סיכומים לקורס מערכות הפעלה

פרופ' יחזקאל ישורון, סמסטר ב' 2009

פרק 1: מבוא:

1.1 הגנה על חומרה:

1.1.1. חשיבות ההגנה על חומרה:

- כתיבת תוכניות למקום אסור בזיכרון , לולאות אינסופיות הגורמות לצריכת משאבי מחשב – מע' ההפעלה מתחמת בעיות אלו ל- *scope* של התוכנית הבעייתית בלבד.
- תוכניות משתמשות במשאבי חומרה בסופו של דבר , מע' ההפעלה מסננת שימוש זה . כך מתאפשרת הרצה בו ז"ז של כמה תוכניות שלא סומכים עליהן לגמרי.

1.1.2. איך מגנים על חומרה:

הגנה על המעבד: למעבד שני מצבים שתמיד נמצא באחד מהם– מצב משתמש (*user*) ומצב מיוחס (*kernel*):

- מצב משתמש – מתאפשרות רק חלק מהפעולות, למשל לא מתאפשרת כתיבה לאוגרים מיוחדים. במצב זה רצות תוכניות רגילות.
- מצב מיוחס – מתאפשרות כל הפעולות, מע' ההפעלה רצה במצב זה.
- מנגנון הפסיקות: מאפשר למע' ההפעלה להשתלט על המעבד מדי כמה זמן. **פסיקה** (*interrupt*) – אירוע חיצוני שלא נגרם ע"י תוכנית, בו המעבד עובר למצב מיוחס ומריץ שגרת פסיקה לפי אוגר הפסיקה (מיוחד). פסיקת שעון – פסיקה המתבצעת כל פרק זמן קבוע כלשהו (מאית שניה). שגרת הפסיקה היא חלק ממע' ההפעלה, יכולה להעביר את המעבד להריץ תוכנית אחרת למשל לכן לתוכניות רגילות אסור לעבוד במצב מיוחס.

הגנה על זיכרון: גישת תוכניות ע"י *load, store*. כל גישה לזיכרון מבוקרת ע"י המעבד, וזאת לפי מבנה נתונים של מע' ההפעלה, שכתובתו באוגר מיוחד (ניתן לשנותו רק במצב מיוחס).

שליטה בהתקנים חיצוניים: שליטה ע"י בקרים (*controllers*). התקשורת בין השני ים נעשית ע"י קריאה וכתיבה ל- *bus*: ערוץ נתונים – עליו הנתון שהמעבד רוצה לקרוא מהבקר או שרוצה לכתוב אליו , ערוץ כתובת – עליו הכתובת שממנה רוצה המעבד לקרוא , ערוץ כתיבה/קריאה – באיזה מצב נמצאים, ופלט/קלט – האם המעבד מתקשר עם הזיכרון או אחד הבקרים . בקר מאזין לכתובות מסוימות, וכשאלה מופיעות על ערוץ הכתובות ב-*bus*, הוא מגיב לכך (ובהתאם לסיבית הפלט/קלט). הנתון יכול להיות נתון לכתוב להתקן או פקודה שעל הבקר לבצע(הזזת ראש דיסק למשל).
תקשורת עם בקר יכולה להיות יזומה ע"י הבקר או התוכנית . התוכנית – כתיבה לכתובות שהבקר מאזין להן, בקר – הדלקת ביט מיוחס שמעביר את המעבד לשגרת פסיקה, שם בד"כ מתחילה התקשורת עם התוכנית. כל דרכי התקשורת מוגנים ע"י מע' ההפעלה – מרחב הכתובות שמאזינים לו בקרים ופקודות הדלקת סיבית הקלט/פלט בפס.

1.1.3. קריאות מערכת והמעבר למצב מיוחס:

קריאות מערכת - *System calls* – קריאות לשגרות של מע' ההפעלה. נקראות ע"י תוכניות כדי שמע' ההפעלה תיגש עבורן לחומרה . בקריאת מערכת מועברים ארגומנטים ומספר מזהה של קריאת המערכת . לפי זהות זו נשלפת ממערך השגרה המתאימה של המערכת . קריאה כזו גורמת לפסיקה יזומה ע"י התוכנית, שכן שגרת המערכת צריכה לרוץ במצב מיוחס. בחזרה מקריאת המערכת משחזר המעבד את מצב המשתמש.

1.2 ממשקים אחידים לחומרה:

למע' ההפעלה יש ספריית שגרות סטנדרטיות לממשק עם התקני חומרה שונים, וכך גישה לחומרה מתבצעת ישירות ללא צורך התערבות המערכת, שכן אלו שגרות של המערכת. ממשקי המערכת מספקים שירות לעתים טוב מהתקני החומרה – למשל מסננות התמודדות עם בעיות אמינות תקשורת בשביל תוכניות, במקום שאלו יצטרכו להתמודד עם זה מול התקן חומרה של קו תקשורת . ישנן יכולות, ספריות שגרות, שלא קשורות לגישה לחומרה, ויכולות להיות חיצוניות למע' ההפעלה. אלו יהיו חלק ממע' ההפעלה משיקולי ביצועים ושיווק הממשקים משתמשים בישויות הבאות:

- תהליך (*process*): מחשב וירטואלי, בעל מעבד אחד או יותר, ריצת תוכנית אחת ע"י משתמש אחד.
- חוט (*thread*): מעבד וירטואלי, יכולים לרוץ וירטואלית בו'ז (ויתכן שגם פיסית).
- קובץ (*file*): דיסק וירטואלי, בעל מאפייני הרשאות, ניתן לכתוב ולקרוא בלוק נתונים בכל אורך שהוא
- קשר (*connection*): מאפשר החלפת מידע בין תהליכים. קשרי רצף (*stream*): העברת רצף נתונים, קשרי מנדעים (*datagrams*): הודעה בדידה.
- חלון (*window*): תצוגה, מקלדת ועכבר וירטואלים. תהליך מציג נתונים בחלון, וכשהוא בפוקוס נתוני מקלדת ועכבר מועברים אליו.
- תור הדפסה (*printer queue*): מייצג מדפסת. תוכניות מדפיסות לתור ומע' ההפעלה שולחת את ההדפסות לפי התור למדפסת הפיסית
- משתמש (*user*): הישות כלפיה מופעלת בקרת הגישה לנתונים ומשאבים. יכול להשתייך ל-*groups*.

1.3 ניהול יעיל והוגן של חומרה:

מע' ההפעלה שולטת בשימוש בחומרה ולכן עליה לנהל זאת באופן יעיל והוגן . יעילות – ניצול מקסימלי של החומרה תוך שימוש מ ינימלי שלה ע"י מע' ההפעלה עצמה – זמינות לת וכניות אחרות. למשל: שימוש מינימלי בזיכרון, זמן מעבד. ניהול הוגן – חלוקת מש אבים בצורה הוגנת אלא אם נאמר מפורשות על תהליך מסוים שחשוב מאחר. שתי סיבות עיקריות לניהול הוגן: 1) לרוב המשתמשים הם בני אדם, ונרצה לתת להם שירות הוגן, כמו מתן שירות ע"י שרת. 2) יצירת תחושה של ריצה בו"ז של מספר תהליכים.

1.4 מערכות הפעלה נפוצות:

שתי קבוצות עיקריות: מערכות יוניקס/לינוקס ומערכות חלונות.

יוניקס: מגוון גדול של מערכות המבוססות על קוד מקור בסיסי , פרט למערכות לינוקס שלא מבססות על קוד יוניקס (פותח עצמאית). ישנם תקנים סטנדרטים לכל מערכות היוניקס, למשל פוסיקס. לינוקס תואמת לו גם כן. הבדלים בין גרסאות יוניקס בממשק משתמש ותוכניתן ותצורת ניהול מערכת המחשב. לגבי לינוקס, חברות שונות מפיצות *distributions* שלהן, שהוא לינוקס בסיסית עם תוספות החברה (הבסיס לא נותן פונקציונאליות רבה למשתמש פרט לממשקים עבור תוכניות חיצוניות).

חלונות: מחולק למערכות *95,98,ME* מבוססות *DOS* ול-*2000,NT*. האחרונות הן מע' הפעלה מודרניות. ישנה אחידות רבה יותר בין גרסאות לעומת יוניקס. הממשק הסטנדרטי של חלונות נקרא *win32*.

מערכות הפעלה נוספות מקטגוריות שונות: מערכות הפעלה ל-*mainframes* המאפשרות הרצת מערכות תוכנה גדולות עם אמינות גבוהה, מערכות זמן אמת, כגון מחשבי מערכות מוטסות טו שליטה למפעלים, מספקות זמן תגובה מובטח, מערכות למחשבים מעוטי יכולת, כגון מחשבי כף יד.

פרק 2: קלט פלט:

2.1 העברת נתונים בעזרת דגימה, פסיקות וגישה ישירה:

ממשק המערכת עם החומרה צריך להיות מהיר ויעיל. יש להתגבר על ממשק לא נכון בין חומרה לתוכנה שגורם אובדן נתונים (למשל תנועת עכבר דורסת תנועה קודמת בחוצץ של הבקר, לפני שהתנועה הקודמת הועברה) ועל חוסר יעילות ותגובה איטית (למשל פירוש פעולות העכבר בעיכוב לפעולה הפיסית).

- דגימה: דרך פשוטה בה דוגמים את התקן הפ"ק (פלט קלט) לעתים קרובות. דגימה תכופה מבזבזת זמן מעבד כי ברוב לא קרה כלום, וכמו כן קשה לדגום התקנים הדורשים טיפול מהיר באירועים שגרת פסיקת שעון, בערך כל מאית שניה, היא דוגמא לכך, אך פרק זמן זה ארוך מדי ולא יעיל.
- פסיקות: טיפול זריז ויעיל באירועים. התקן מפעיל פסיקה והמעבד עובר לשגרת פסי קה המבצעת טיפול בהתקן. הטיפול יעיל – רק כאשר ההתקן נדרש לכך, ומהיר – לרוב מתעכב רק ע"י הזמן הלוקח למעבד לאחסן את המצב הנוכחי שיוכל לשחזרו לאחר הטיפול בהתקן. חסרון: חוסר יעילות כשצריך קצב העברת נתונים גבוה מאל התקן בשל עלות גבוהה של "שיחת" מעבד-התקן. פתרון:
- גישה ישירה לזיכרון (*DMA*): המעבד מעביר לבקר להיכן להעביר /לקבל את הנתונים בזיכרון. ניהול ההעברה (השיח) מתבצע ישירות בין הבקר לזיכרון, במקום בין הבקר למעבד. חיסרון: צורך בקר מתוחכם שלרוב יהיה יקר יותר. פתרון:
- בקר גישה ישירה: שיתוף בקר גישה ישירה בין כמה בקרים. המעבד מורה לבקר זה פקודות עבור בקרים אחרים, והם מתקשרים דרכו עם הזיכרון.

2.2 עוד אודות פסיקות:

המעבד יכול להבחין כאשר מקבל פסיקה, אך לא מאיזה בקר קיבל אותה. בקר פסיקות – דרכו מועברות פסיקות כל שאר הבקרים למעבד. בעת הפעלת פסיקה, המעבד לוקח מבקר הפסיקות (היחיד) את זהות המקור, שהוא הבקר המקורי שקרא לה, ומופעלת השגרה המתאימה. שגרת פסיקה לא תרוץ פעמיים במקביל, אלא בתור, על כן צריכות להיות מהירות. אם יש פעולה ארוכה, היא תתבצע לאחר סיום שגרת הפסיקה ולא כחלק ממנה, באופן הבא: שגרת הפסיקה מכינה מבנה נתונים המייצג פעולה דוחיה של שגרה, שיחזיק כתובת שגרה וארגומנטים פחות דחופים. שגרת הפסיקה תסיים את הפעולות הדחופות, ומבנה הנתונים יוחזק בתור שגרות דחויות לאחר סיום שגרות הפסיקה יתבצעו אחרות אם יש, ורק אח"כ השגרות הדחויות.
מע' הפעלה מסוימות מסווגות פסיקות ע"פ עדיפויות. פסיקה בכניסה נמוכה בבקר הכניסות – דחיפות גבוהה; פסיקה גבוהה – דחיפות נמוכה. אם בעת שגרת פסיקה נכנסה פסיקה מעדיפות גבוהה יותר– הנוכחית תופסק. אם לא, היא תפעל עד סיומה.

2.3 ממשקי תוכנה/חומרה: מנהלי התקן:

תקשורת עם התקן חומרה מסוים נעשה ע"י תוכנה הנקראת מנהל התקן (*device driver*). לכל התקן המחובר לבקר יש מנהל התקן.

2.3.1. הממשק הסטנדרטי למנהלי ההתקן:

בלינוקס הממשק הסטנדרטי כולל את השגרות הבאות:

- *Read, write*: קריאה וכתיבה, יכול להעשות דרך חוצץ ליעול
- *Poll*: דגימה, לא ידובר.
- העברת פקודות אחרות שלא ניתנות להעברה דרך הממשק הסטנדרטי, גישה לתכונות מיוחדות של ההתקן.
- איתחול ההתקן: בודקת האם קיים, ואם כן מאתחלת אותו.
- *Open, close*: התחברות והתנתקות תהליכים להתקן. מטפלת במקרה בו שני תהליכים מנסים לגשת להתקן, או שתהליך מתנתק מההתקן.
- *Flush*: כתיבת נתונים שמצטברים בחוצץ להתקן הפיסי.

מנהלי ההתקן כוללים גם שגרות פסיקה, שנקראות כשהתקן יוזם תקשורת עם מנהל ההתקן ע"י הפעלת פסיקה.

2.3.2. מנהלי התקן עם ובלי הטמנה:

הבחנה בין מנהלי התקן להתקני זיכרון – המשתמשים במאגר החוצצים המרכזי של מע ' ההפעלה, ובין אחרים. הראשונים קרויים מנהלי התקן עם הטמנה (*block devices*) והאחרונים *character devices*. הראשונים מופעלים ע"י מאגר החוצצים המרכזי (*buffer cache*) של המערכת. מחזיק נתונים אחרונים, ובהם יעשה שימוש ללא קריאה למנהל ההתקן. קריאה תעשה כשהנתונים לא בחוצץ או לצורכי כתיבה לפינוי החוצץ.

האחרונים מופעלים ללא מאגר החוצצים, כמו מקלדת ועכבר. יכול להיעשות שימוש בחוצץ שלא ממאגר החוצצים, אך לצורכי גישור פערי מהירות ולא חיסכון גישה להתקן. נעשה שימוש בשיטה זו גם לדיסקים אם לא נדרשת חציצה של המערכת למשל: *raw access* – גישה ישירה לדיסק.

2.3.3. התייחסות להתקנים ומנהלי התקן ביוניקס ולינוקס:

גישה להתקני חומרה במערכות אלו כמו גישה לקבצים. למשל: גישה ל-*/dev/cdrom*. הקישור של המערכת של שמות אלו להתקנים נעשית ע"י החזקת שתי טבלאות: אחת למנהלי התקן עם הטמנה והשניה לאלו בלי. שגרת האתחול של מנהל התקן שמה במקום המתאים בטבלה, לרוב נקבע מראש בבניית מערכת ההפעלה, מצביע למבנה הנתונים של מנהל ההתקן המחזיק שגרות של מנהל ההתקן (כמו *open* לדיסק). המיקום בטבלה קרוי המספר הראשי (*major number*) של ההתקן/מנהל ההתקן. אם שולט בכמה התקנים, גישה תעשה דרך מספר משני (*minor*).

Special file: קישור בין שם כמו לעיל למנהל התקן (לא קובץ!), ע"י ציון ההתקן ע "י מספרו הראשי והמשני. יתכן קיום קובץ מיוחד ללא קיום מנהל ההתקן עצמו (סה"כ קישור). יתכן גם מנהל התקן שאין לו קובץ מיוחד , ואז לא תתכן גישה אליו מתוכניות רגילות (המערכת יכולה לגשת ע "י ציון המספרים שלו ישירות). *Device file system (devfs)*: שיטת יצירת קבצים מיוחדים אוטומטית למנהלי התקן שקיימים במערכת

2.3.4. העברת פקודות מיוחדות להתקנים: -

2.4 דיסקים קבועים ותזמונם:

דיסק מגנטי: מורכב ממספר צלחות (*plates*) על גבי ציר משותף. לכל צלחת ראש קורא/כותב היושב על זרוע הנעה פנימה והחוצה. הזזת ראש (*seek*) לוקחת כמה אלפיות שניה. הנתונים יושבים על מסילות (*tracks*) קונצנטריות, כל אחת מחולקת לקטעים (*sectors*) בגודל קבוע (512-4096 בתים). נתונים עוברים מהר יותר במסילות חיצוניות מאשר בפנימיות (מהירות גבוהה יותר משוליים). נתונים עוברים בקטעים שלמים. קבוצת כל המסילות זו על זו בכל המשטחים נקראת גליל (*cylinder*). העברת נתונים: הזזת הראש לגליל המתאים וסיבוב הדיסק למיקום המתאים. הזזת הראש לוקחת הזמן הרב ביותר. לכן שואפים לסדר נתונים באותו גליל/גלילים סמוכים.

בקרים פשוטים יכולים לקרוא קטע אחד ב כל פעם, ולכן סירוג (*interleaving*) הנתונים על הדיסק (לא ברצף) מחפה על הזמן שבין סוף קריאת קטע ראשון ועד תחילת קריאת הקטע הבא. בקרים מתוחכמים קוראים מסילה שלמה לחוצץ או יכולים לקרוא כמה קטעים רציפים. המערכת לרוב צריכה לסדר את הבקשות הרבות לדיסק לפי אלגוריתם שצריך להשיג:

- הגינות: קדימות בקשות מוקדמות על פני מאוחרות.
- מניעת הרעבה: אסור שתהיה בקשה שלא תקבל מענה אף פעם. אמנם אין הגינות מושלמת, אך אסור להגיע להרעבה לעולם.
- יעילות: הקטנת זמן ההמתנה של הבקר להזזת הזרוע או סיבוב הדיסק.

אפשרויות:

- *FIFO*: לפי סדר קבלת הבקשות, הוגנת ומונעת הרעבה אך עלולה להיות לא יעילה ביותר(תקורה גבוהה מאוד).
- *SSTF – shortest seek time first*: בחירת הבקשה שניתן למלא תוך המתנה מזערית להזזת הראש וסיבוב הדיסק. נפסלת כי יכולה להרעיב
- שיטת מעלית: איזון בין השתיים לעיל. הפשוטה ביותר, *scan*, הזרוע נעה פנימה והחוצה במחזוריות כך ששהבקשה הקרובה ביותר לכיוון התנועה היא שנענית. בהגעה לגליל מסוים, כל הבקשות בתור (לא חדשות) באותו גליל נענות. *C-scan*: דומה, יעילה פחות ויותר הוגנת, שירות רק בתנועה פנימה ולא בתנועה החוצה. וריאציות יעילות יותר: *look, c-look*: הזרוע נעה רק עד הגליל הקיצוני ביותר עליו יש בקשות ולא עד הסוף.

2.5 דיסקים לוגיים ומערכי דיסקים:

2.5.1. מחיצות:

חלוקת דיסק לדיסקים לוגיים בשם *partitions*. לכל מחיצה גלילים משלה , ומערכת ההפעלה מתייחסת אליהם כדיסקים נפרדים. טבלת המחיצות נשמרת בקטע מיוחד בתחילת הדיסק. התוכנה *Fdisk* מאפשרת לשנותה.

2.5.2. דיסקים לוגיים:

LVM – logical volume management, שיטה ביוניקס/לינוקס בה כל הדיסקים מחולקים לקטעים בגודל קבוע בשם *extens*, ומהם ניתן להרכיב *logical volume* – דיסק לוגי. יתרונות על מחיצות: 1) שינוי גודל ביתר קלות (לא חייבות רציפות), שינוי מבנה הנתונים של ארגון הקבצים בלבד. 2) הרכבת דיסק לוגי מקטעים על גבי כמה דיסקים פיסיים.

2.5.3. מערכי דיסקים:

RAID : שימוש במספר דיסקים כבהתקן אחסון אחד. מספקים שירותים שונים שלא ניתנים ע"י דיסקים לוגיים. מתחלק לרמות שונות:

- *RAID 0* : הבלוקים שמורים על הדיסקים באופן בלוק מחזורי . כלומר, הבלוק ה-i יהיה בדיסק ה- $i \bmod \#disks$, כבלוק מספר $i \operatorname{div} \#disks$. שיטה זו נקראת *striping* – שמירה ברצועות. רצועה היא קבוצה של d בלוקים רצופים השמורים על כל d הדיסקים. גישה מהירה לכל הדיסקים יחד, כל זמן שהמערכת ניגשת לבלוק שונה מכל דיסק, כשנקראים רצפים ארוכים של נתונים הרצף מחולק באופן שוה בערך בין כל הדיסקים
- *RAID 1* : הבטחת אמינות, לא מהירות. כל בלוק נשמר על שני דיסקים או יותר כגיבוי– *mirroring*.
- *RAID 3* : כמו רמה 0 רק שאחד הדיסקים שומר *parity* של הנתונים על שאר הדיסקים כדי לספק יתירות (*redundancy*). הבדל נוסף הוא שגישה נעשית בכל הדיסקים לאותו בלוק , ואי אפשר לבלוקים שונים על דיסקים שונים , וזאת כדי שניתן יהיה לכתוב רצועה שלמה בגישה אחת , אחרת היה צורך שינוי שני בלוקים בדיסק הזוגיות בבת אחת . הכתיבה המרובה לדיסק ה זוגיות היה מעכב את הכתיבה בכלל . ע"י הזוגיות ניתן לשחזר נתוני דיסק אחד , אם חסר או לא תקין , בכל רצועה. יתרונות: קצב העברה גבוה לכמה דיסקים יחד , זוגיות; חסרונות: מחיר דיסק נוסף לזוגיות וגישה לרצועה שלמה בלבד ולא לבלוקים שונים בדיסקים שונים מתאים לרצפים ארוכים.
- *RAID 5* : מאפשר גישה לבלוקים בודדים ולא רק לרצועות שלמות. בלוק הזוגיות של כל רצועה i שמור על הדיסק ה-i. כך ניתן להתגבר על הבעיה מרמה 3 ולכן ניתן לכתוב לשני בלוקים מרצועות שונות על דיסקים שונים במקביל (בתנאי שהכתיבה היא לא לדיסקים המחזיקים את בלוקי הזוגיות אחד של השני). חסרון מרמה 3 : כל כתיבה מצריכה תחילה קריאה מהבלוק ומבלוק הזוגיות שלו, וב-3 אין צורך כלל בקריאה מוקדמת.

מערכות הפעלה שונות מסוגלות לממש מערכי דיסקים אלו בדיסקים ובקרים רגילים, אך יש בקרים מיוחדים שחוסכים זאת למע' ההפעלה. מי שמטפל במערך (המערכת או הבקר) צריך לטפל בפיזור הנתונים, זוגיות וטיפול במקרה תקלה : כתיבת נתונים בדיסק רזרבי או שחזור נתונים בדיסק שמוחלף , תוך כדי עבודה רצופה על המערכת. יתירות דיסקים אינה מספיקה לעבודה רצופה, כי גם התקנים אחרים עלולים להתקלקל: ספקי כוח, בקרים וכו'.

<u>2.6 התקני פ"ק נוספים</u>:

2.6.1. דיסקים אופטיים:

<u>תקליטורים</u> (*CD*) : מידע מקודד ע"י חורים במשטח, מזוהה ע"י ליזר אם יש חור או לא , עד *MB*700. *DVD* : שומרים עד *GB*17. יתרונות: 1) הפצת מידע דיגיטלי זולה, אחסון זול ועמידות לאורך שנים.

<u>סרטים מגנטיים</u> : חומר מגנטי עוטף סרט פלסטי השמור לר וב בקלטת , גישה איטית אך יכולה להכיל כמה עשרות *GB*. שימוש בעיקר לארכיונים , והקריאה בכוננים מיוחדים לרוב כחלק מספריה רובוטית . סוגים: סרט סלילי , הנכרך על ראש קורא /כותב ומסתובב במהירות , וסרט לינארי , העובר בסמוך לראש, אורך חיים גבוה שכן אין שחיקה של הסרט/הראש מהמגע. מדיה זולה מדיסקים, וניתן לשמור מידע חדש שוב ושוב

<u>פרק 3: ניהול זיכרון</u>:

מנגנון ניהול הזיכרון יוצר 3 יכולות חשובות:

- הרצה בו"ז של כמה תוכניות תוך הגנת זיכרון כל תוכנית משאר התוכניות
- שימוש בדיסקים כהרחבה של הזיכרון ע "י ניהול של המערכת . זמן גישה לדיסק גבו ה בהרבה מלזיכרון הראשי . השמת נתונים שהגישה אליהם נדירה על הדיסק, וכאלה שהגישה אליהם תכופה על הזיכרון, שומרת על זמן גישה ממוצע טוב.
- הזזת מבני הנתונים של תוכנית בזיכרון תוך כדי ריצה שלה

הרעיון העיקרי שמאפשר זאת: הפרדה בין כתובות בתוכנית לכתובות הפיסיות – זיכרון וירטואלי.

<u>3.1 זיכרון וירטואלי</u>:

לכל תהליך מרחב זיכרון וירטואלי בגודל הזיכרון שניתן להתייחס אליו בפקודה – 32 ביט, כלומר גודל הזיכרון $2^{32} = 4GB$. במחשבי 64 ביט זה גדל בהתאם. זה מתחלק לבלוקים בגודל קבוע הנקראים <u>דפים</u>, לרוב בגודל *4KB*. המערכת מקצה לתהליך דפים לפני הרצת תוכנית – קוד התוכנית, מחסנית, קבועים. במהלך ריצה, ע"י קריאות מערכת, המערכת יכולה להקצות לתהליך עוד דפים.

הזיכרון הפיסי מחולק ל- *frames* בגודל דפים, המאחסנות דפי תהליכים. מבנה נתונים מיוחד ממפה דפים למסגרות . כדי לאפשר הקצאת יותר דפים מכמות הזיכרון הפיסי , המערכת מאחסנת דפים באזור בדיסק הקרוי <u>איזור הדפדוף</u> (*backing store*), המחולק למסגרות בגודל דף . תוכנית רצה בהתייחסות למרחב הכתובות הוירטואלי שלה . המערכת יחד עם החומרה מתרגמת לכתובות פיסיות . <u>חריג דף</u> (*page fault*) הוא ניסיון גישה לדף שממופה לדיסק ולא לזיכרון הפיסי, מטופל ע"י המערכת.

<u>3.2 תרגום כתובות וירטואליות לפיסיות</u>:

מתבצע לרוב ע "י *TLB*, חלק מהמעבד . טבלה המכילה תרגומים של דפים למסגרות בזיכרון . כתובת וירטואלית מפורקת לשני חלקים : 1) מספר דף (*MSB*) 2) היסט (*offset*) מתחילת הדף (*LSB*) – גודל נקבע על פי גודל הדף. למשל לדף בגודל $4KB = 2^{12}B$ צריך 12 ביטים לתיאור היסט. לאחר פירוק נעשה תרגום למספר הדף ב-*TLB* ומוחזר מסגרת פיסית. זה משורשר להיסט ומתקבלת הכתובת הפיסית. אם הדף לא ב-*TLB*, זהו חריג *TLB*.

במקרה של חריג *TLB* מחפש המעבד את המיפוי ב-*page table* בזיכרון הראשי, ומכילה מיפויים לכל הדפים למסגרות פיסיות. החיפוש יחל ב-*TLB* שכן הוא מהיר יותר, ורק אם היה חריג *TLB* תעשה גישה לטבלת הדפים בזיכרון. לאחר שליפה מהזיכרון, לרוב יוכנס המיפוי ל-*TLB*.

3.3 מבנה טבלאות דפים:

צריך להיות פשוט לחיפוש ע"י מנגנון חומרה, לאפשר חיפוש במספר מינימלי של גישות לזיכרון ולצרוך מעט מקום סוגים:

- טבלת דפים שטוחה (*flat*): מערך המכיל את כל רשומות המיפוי של מרחב זיכרון וירטואלי חיסרון: תופסת מקום רב בכל מקרה.
- טבלת דפים היררכית דלילה (*hierarchial*): שמירת מיפוי בעץ חיפוש רדוד . בכל רשומת מיפוי במערך העלים יש ביט *valid* המעיד על תקפות הרשומה. דליל כיוון שלא צורך זיכרון לתחומי דפים שלא מוקצים (תת עץ מתאים פשוט יהיה מאופס). זהו היתרון על טבלת דפים שטוחה. חסרון הוא (מלבד תפיסת יותר זכרון במקרה שכל התחום מוקצה) מספר גישות לזכרון גבוה יותר – כעומק העץ, ושטוחה – בכל מקרה גישה אחת. מנגנון החיפוש מורכב יותר מטבלה שטוחה.
- טבלת דפים הפוכה (*inverted*): שמירת מיפוי ל כל ה דפים הממופים למסגרת פיסית ע "י טבלת *hash*. כל רשומה שומרת מספר מסגרת פיסית , מספר התהליך (מזהה מרחב הזיכרון הויר') ומספר הדף במרחב הויר'. הוצאת רשומה ע"י חישוב פונ' *hash* על מס' הדף. אם יש התאמה למספר הדף ולמזהה התהליך , מוחזר הערך. יתרונות: 1) גודל תלוי בגודל הזכרון הפיסי ולא מס ' התהליכים הרצים בו"ז. 2) טבלת *hash* מצריכה מספר נמוך של גישות לזכרון בממוצע החסרון: מורכבות גדולה יחסית של מנגנון החיפוש.

3.4 תחזוקת טבלאות הדפים וה-*TLB*:

המערכת אחראית על תחזוקת טבלאות דפים. בהעברת דף ממסגרת אחת לאחרת, מעדכנת את טבלת הדפים המתאימה. תחזוקת ה-*TLB* נעשית בשיתוף המערכת עם המעבד . המעבד יכול לטפל בחריג *TLB* בעצמו או ע "י שגרת פסיקה מיוחדת , והמערכת תטפל בו ותעדכן את ה- *TLB* ע"י פקודת מכ ונה מיוחדת. חסרון: יקר יותר מטיפול ע"י חומרה. יתרון: מפשט את תכנון המעבד, חופש במבנה הנתונים לטבלאות הדפים. לרוב מטופל במנגנון חומרה. כך או כך המערכת היא האחראית למחוק רשומות לא רלוונטיות מה-*TLB* (למשל בהעברה למסגרת אחרת או לדיסק). לכך יש פקודות מכונה מיוחדות. המערכת אחראית להודיע למעבד מהו מרחב הזיכרון הוירטואלי לתהליך שרץ כעת . בטבלאות שטוחות/היררכיות המערכת נותנת את כתובת טבלת הדפים, ובהפוכה המערכת מודיעה על מזהה מרחב הזכרון הויר של התהליך שרץ. גישה ל-*TLB* ולזכרון טבלאות המיפוי נעשה רק במצב מיוחס.

3.5 הגנה על זיכרון וירטואלי:

תהליך התרגום מהווה מנגנון הגנה על הזיכרון. שני חלקים עיקריים: 1) בתרגום לא ניתן כלל להתייחס למסגרות פיסיות שלא מוקצות לתהליך הנוכחי. 2) כל מיפוי מכיל בקרת גישה, הנקראת ע"י המעבד וקובעת הרשאות גישה – מצב משתמש ומיוחס או רק מיוחס (לקריאה, כתיבה, ביצוע). כך המערכת יכולה להקצות מבני נתונים שלה במרחבי זיכרון של תהליכים בלי חשש ל גישה לא מורשית. כך בעת פסיקה או קריאת מערכת לא צריך להחליף את מרחב הזיכרון הויר' הנוכחי. כל ניסיון גישה לא חוקי מפעיל שגרת פסיקה של המערכת, כאשר: דף וירי לא מוקצה, לא ממופה למסגרת פיסית או ממופה ללא הרשאות. אלו חריגי דף – *page/segmentation fault*.

3.6 טיפול בחריגי דף:

דף שאינו ממופה במצב רגיל: ניסיון גישה לדף שלא ממופה למסגרת פיסית . המערכת בודקת: אם ניסיון הגישה לדף שלא הוקצה , התוכנית עפה. אם הדף מוקצה אך למסגרת בדיסק , המערכת תביא את המסגרת לזיכרון ותעדכן את טבלת הדפים , תו"כ שתהליכים אחרים ממשיכים בינתיים לרוץ . בגישה הבאה לא יהיה חריג. אפשרות נוספת: הדף לא ממופה לאף מסגרת, כאשר זהו השימוש הראשון בדף. המערכת תביא את המסגרת לזיכרון.

דף ממופה עם הרשאות שאינן מספיקות במצב רגיל: המערכת מעיפה את התוכנית או מודיעה על חריג . מגן על דפים של המערכת או ניסיונות גישה של דפים מוגנים חלקית. עם זאת לא מגן על המערכת או על תוכנית אחת מפני אחרת.

דף שאינו ממופה במצב מיוחס: חריג הנובע מדף הממופה למסגרת בדיסק, דף שטרם נעשה בו שימוש ולכן לא ממופה כלל או שגיאת תכנות. בראשונים המערכת מקצה וממפה . באחרון המערכת צריכה להבחין בין שגיאה של התוכנית ובין שגיאה שלה . אם זוהה שהועבר ארגומנט שגוי לקריאת מערכת , התוכנית עפה/מוחזר חריג, אם לא – זוהי שגיאת מערכת, ומטופלת ע"י הפסקת פעולת המערכת או אתחול תת המערכת בה קרתה התקלה.

דף ממופה עם הרשאות שאינן מספיקות במצב מיוחד: לרוב נובעת מתקלה במערכת.

בעת הבאת מסגרת מהדיסק לזיכרון , המערכת תביא כמה מסגרות (*prefetching*) למספר דפים עוקבים בזיכרון הויר'. לכן צריכה לנסות לשמור דפים רצופים במסגרו תרצופות בדיסק. מצד שני עלול לפגוע בביצועים: 1) אם המסגרות לא עוקבות בדיסק , הבאת רצף דפים יהיה יקר . 2) אם התוכנית לא תשתמש באקסטרות מהר, הם הובאו לשווא, ויתכן שבגללן פונו מסגרות אחרות בזיכרון שיגרמו בעתיד לחריגי דף. המערכת יכולה להגביל *prefetching* למקרים בהם למשל היה רצף של כמה חריגי דף כתוצאה מניסיון גישות לדפים רצופים(מעיד למשל על גישות למערך רצוף).

3.7 התייחסות לכתובות פיסיות:

המערכת צריכה להתייחס לפעמים לכתובות הפיסיות , למשל בהעברת פקודות לבקר *DMA* – צריכה להעביר לו כתובת פיסית של חוצץ (יש בקרים שיכולים לבקש את התרגום ישירות מהמעבד). כמו כן צריכה לודא שבקר מעביר נתונים למסגרות רצופות בזכרון הפיסי? דוגמא נוספת: העברת כתובת

טבלאות הדפים למעבד. חלק מהמערכות מקצות חלק ממרחב הזכרון הויר' להתייחסות לזכרון פיסי.

3.8 מנגנונים ומדיניות לפינוי מסגרות:

פינ וי דפים אורך זמן: מציאת דפים מתאימים לפינוי (קורבנות) וכתיבת תוכנם לדיסק. עם זאת יש צורך בהקצאה מהירה כדי לשמור על זמן תגובה טוב של תוכניות ושל המערכת עצמה, לכן יש צורך לשמור על מלאי מסגרות פנויות בזיכרון הפיסי. מתבצע על ידי שני מנגנונים: 1) שגרות של המערכת הקורות כל פרק זמן קבוע ומפנות מסגרות. 2) בעת חירום מנסה לפנות מס' מינימלי של מסגרות בעת חריג דף כשהמלאי קטן.

המנגנון הראשון מורכב משני מבני נתונים ושלוש שגרות עיקריות: מבנה אחד מחזיק מסגרות מועמדות לפינוי שצריך לכתוב לדיסק. מסגרות ששונו בזיכרון מאז שנכתבו מהדיסק קרויות מסגרות מלוכלכות (*dirty*). השני מכיל מסגרות נקיות (לא צריך לכתבן לדיסק). השגרות: 1) מוצאת מועמדות לפינוי, מוחקת אותן מטבלת הדפים ושמה במבנה המתאים (מלוכלכת/נקיה). 2) מעתיקה מסגרות מלוכלכות לדיסק ומנקה אותן. תורמת להגדלת מלאי מסגרות מהירות לפינוי (נקיות) ולא משפיעה לרעה על ביצועים כי לא כותבת לדיסק בזמן שתוכנית מנסה להשתמש בו. 3) פינוי סופי של מסגרות נקיות ע"י מחיקת תוכנן, לא חיונית כמו השתיים הראשונות. כל הפעלה תפנה מס' מסגרות בהתאם למדיניות מע' ההפעלה. ככל שיש יותר מסגרות פנויות, כך יפונו פחות אם בכלל. אי פינוי מיידי של מסגרות נקיות מאפשר שימוש בהן (שחזור מהיר בעת צורך) עדיין תוך שדהן במאגר מועמדות לפינוי.

שיטות החלטת סדר פינוי: *least recently used – LRU*: פינוי המסגרת שהשימוש האחרון בה הכי רחוק בעבר. מצריך חותמת זמן בכל דף בכל שימוש בו. ב-*TLB* לרשומות יש ביט *used* שדולק בעת שימוש בדף, ומועתק לטבלת הדפים בעת מחיקת הרשומה מה-*TLB*, וזהו קירוב ל-*LRU*. כמו כן פקודת מכונה מיוחדת מעדכנת את טבלת הדפים. הביט הזה מאופס כל פרק זמן קבוע. קורבנות עם ביט דולק יבחרו אחרונים לפינוי. מערכות המשתמשות ב-*LRU* בחורות מסגרות לפינוי בשני שלבים: 1) בחירת תהליך ממנו דפים יפונו. 2) בחירת מסגרות ספציפיות. לינוקס – בחירת התהליך שגרם למינימום חריגי דף לאחרונה. חלונות – *working set*: הערכה של המערכת למספר המסגרות שתהליך צריך, נקבע לפי קצב חריגי הדף שלו. לתהליך שגורם הרבה חריגי דף יוגדל ה-*working set*, וההיפך. הגדלה/הקטנה זו חסומה.

יתרונות גישת שני השלבים על פני התייחסות שווה לכל הדפים של כל התהליכים: 1) מימוש הגינות ביחס לתהליכים שרצים בו"ז. בהתייחסות שווה לכל הדפים, יתכן יתכן שיפונה דף מתהליך דל דפים ודף מתהליך מרובה דפים – חוסר הגינות. 2) חיפוש מסגרות לפינוי יכול להי ות לא יעיל, שכן מצריך מעבר על כל טבלאות הדפים (לא יעיל אלא בטבלת דפים הפוכה). לעומת זאת, לפי שיטת שני השלבים קודם נבחר תהליך ונעשה חיפוש רק בטבלה שלו.

3.9 איזורי דפדוף ומיפוי קבצים:

שני סוגים: 1) למסגרות אנונימיות שהוקצו במרחבי זיכרון ויר'. אזור כזה יכול ל השתמש בדיסק, למשל בחלונות: *c:\pagefile.sys*. בלינוקס ניתן להשתמש בכמה אזורי דפדוף והגדרת עדיפויות (למשל העברה לאזור דפדוף בדיסק מהיר). 2) משמש למיפוי קבצים למרחב הזיכרון הויר'. תוכנית מבקשת ע"י קריאת מערכת מיוחדת דפים במרחב שימופו לקובץ מסוים, וכתיבה למרחב זה תפורש בעת עדכון בדיסק לכתיבה לקובץ. כך הקובץ הנמצא בזכרון הפיסי, גם טרם עדכונו בדיסק, יכול להיות משותף לתהליכים אחרים מיד עם סיום הכתיבה. שימוש עיקרי במיפוי קבצים: טעינת קוד לזיכרון. יתרונות על הקצאת מסגרות אנונ': 1) מיפוי זה תופס מקום בדיסק, אך ממילא ה קובץ נמצא על הדיסק, וכך נחסך מקום. 2) לא כל קוד התוכנית מתבצע, וכך מה שלא מתבצע – לא נטען מלכתחילה לזכרון.

3.10 שיתוף זיכרון בין תהליכים ו-*dynamically linked libraries (DLL)*:

מנגנון הזכרון הויר' מאפשר מיפוי קוד ומבני נתונים למספר מרחבי זכרון וירטואלים שונים. סיבות לכך: מספר תוכניות הרצות בו"ז משתמשות באותן שגרות, כגון *printf* של תוכניות *c*. כך נמנע שכפול מידע במסגרות פיסיות בזכרון. הספריה בנויה כך שההתייחסויות לכתובות יחסיות למיקום השגרה (*PIC*). ספריות אלו קרויות *DLL* או *SO* בלינוקס/יוניקס. בבנית התוכנית הלינקר לא מוסיף לקוד הבינארי את השגרות אלא התייחסות סימבולית ל-*DLL* ולשם השגרה. בעת טעינת תוכנית תמופה הספריה למרחב הזכרון הוירטואלי שלה, ואם יש כמה תהליכים, לכל היותר יהיה עותק אחד שלה במסגרות הפיסיות.

סיבה נוספת: העברת נתונים ביעילות בין תוכניות. הדרך היעילה ביותר להעברת מידע בין תהליכים (יעילה מאשר דרך קבצים). אמנם תהליכים פגיעים כך לשגיאות תהליכים אחרים בשימוש בזיכרון המשותף, אך פחות מאשר אם לכל תהליך גישה לזכרון של כל תהליך אחר. ניתן לחזק את ההגנה ע"י הרשאות מתאימות לתהליכים שונים למרחב הזכרון המשותף (למשל הורדת הרשאה לקריאה בלבד כשאפשר).

פרק 4: תזמון מעבדים ומיתוג תהליכים:

4.1 תהליכים ומיתוג תהליכים:

תהליך במערכת מיוצג ע"י מבנה נתונים עם המרכיבים הבאים:

- מצב המעבד: תוכן האוגרים למעבד, אוגרי נתונים, מצביע התוכנית ומצביע המחסנית, חלק מהאוגרים המיוחדים. נשמר בהפסקה, לשחזור אח"כ.
- מפת זכרון: ממפה את מרחב הזכרון הויר' של התהליך לזכרון הפיסי ולאזורי הדפדוף.
- טבלת שגרות איתות: שגרות לטיפול פסיקות ויר' (אתות), למשל אירועי פ"ק. אם תהליך מושהה, בשחזור יש להגיב בהתאם (אירוע קרה בהשהיה).
- זהות והרשאות: הרשאות תהליך תלויות המשתמש. המערכת זוכרת את השראות המשתמש והשינוי לעומת הרשאות המערכת עצמה.

- רשימת משאבים זמינים: המערכת מחזיקה רשימת משאבים שהתהליך ביקש וקיבל גישה, וסוג ההרשאה שקיבל. בעת בקשה לגישה לקריאה / כתיבה (*open*) וכו' מכניסה המערכת מצביע למשאב לרשימה זו ומחזירה מזהה משאב (*handle*), המצביע על מקום זה ברשימה. האיבר ברשימה מצביע למבנה נתונים של המערכת המתאר את המשאב. בעת *close* נמחק האיבר מהרשימה. סיבות למנגנון: 1) הרשאות שמורות בטבלה, לא צריך לבדוק בכל גישה למשאב. 2) הצבעה לא ישירה נותנת שליטה על המשאב למערכת. בחלונות אין הבדל בגישה כך לקבצים, התקני פ"ק ותהליכים.
- מצב תזמון וסטט' שונות: המערכת זוכרת לתהליך מושהה הדוע מושהה: האם אפשר להחזירו או שממתין לאירוע. שומרת נתונים על פעילות עבר כדי להחליט לגבי תזמון התהליך והזכרון הוי' שלו. למשל החזרת תהליך לרוץ על אותו מעבד, הקצאת זכרון נוסף עקב קצב חריגי דף גבוה.

תהליך (*process*): מחשב וירטואלי עם כל המרכיבים. חוט (*thread*): מעבד וירטואלי מתוך כמה בתהליך. חוטים בתהליך משתפים זכרון, הרשאות, רשימת משאבים זמינים, מצב האיתותים. הפרטי לכל חוט: מצב המעבד ומצב התזמון של החוט. מצב התזמון מחזיק נתונים גם לכל חוט בנפרד.

4.2 מיתוג תהליכים:

Contest switching – מיתוג: לפני המיתוג מעבד ממופה לחוט של תהליך מסוים, מריץ את פקודות תוכנית החוט עם מפת הזכרון של התהליך. שלבים:

1. מעבר לשגרה של המערכת תוך שמירת המצב במחסנית (לשחזור). המעבר קורה מבקשת שירות של המערכת או פסיקה אחרת שגורמת לשגרת פסיקה: חריג דף, פסיקת בקר לפ"ק, פסיקת שעון.
2. המערכת מטפלת באירוע שגרם לשגרת הפסיקה.
3. המערכת מחליטה שהמעבד צריך להריץ חוט/תהליך אחר לפי מדיניות כלשהי.
4. המערכת שומרת את מצב החוט/תהליך הקודם במבנה נתונים של המערכת.
5. אם המעבר לתהליך עם מפת זכרון שונה, המערכת משנה את טבלת הדפים למעבד.
6. המערכת משחזרת את מצב המעבד של החוט (מעבד וירי') שצריך לרוץ ומריצה אותו.

4.3 תהליכי וחוטי גרעין:

מערכות משתמשות בתהליכי/חוטי גרעין (*kernel process/thread*) כדי לממש מנגנונים שקורים כל פרק זמן מסוים, כמו פינוי דפים. כיון שמשתמשים במבנים של המערכת שממופים למרחבי הזכרון הויר', אין להם צורך במרחב זכרון. כשרץ תהליך כזה ללא מפת זכרון, לא מוחלפת מפת הזכרון של המעבד: חסכון עלות ההחלפה. שימוש בתהליכי גרעין כאשר יש רצון בתחרות עם תהליכים רגילים, להם יותר עדיפות מתהליכי הגרעין (כמו מנגנון ניקוי מסגרות מלוכלכות, בעדיפות נמוכה). כששום דבר לא רץ ברירת המחדל היא *system idle process*, תהליך שלא עושה כלום. שדונים (*deamons*) – תהליכים של המערכת שכן דורשים זכרון ויר' פרטי.

4.4 יצירת תהליכים בלינוקס ויוניקס:

יוניקס ולינוקס משתפוץ זיכרון בין תהליכים בצורה יעילה. קריאת המערכת *Fork* היוצרת תהליך חדש משכפלת תהליך קיים. מוחזרים שני התהליכים (מקורי ומשוכפל) זהים בכל, אך נפרדים: שינוי ערכים באחד לא משפיע על השנ', כנ"ל פתיחת/סגירת משאבים. למעשה, פיצול תהליך.

דרך אחת ליצירת תהליך חדש: העתקת כל המ סגרות של התהליך הקיים, כך מרחב הזכרון זהה אך נפרד. לא יעיל כי צריך לחכות עד שההעתקה תסתיים עד לחזרת פקודת ה-*fork*, וחלק גדול של המסגרות לא ישונו יותר והועתקו לחינם (ניתן היה לשתפם). דרך נוספת, של יוניקס ולינוקס, שימוש במנגנון *copy on write*: שימוש באותו זכרון עד שאחד מנסה לשנות מסגרת – ואז מועתק לו עותק פרטי שלה. פועל כך: בעת *fork* משתנות הרשאות הגישה למסגרות משותפות לקריאה בלבד. טבלת הדפים של התהליך מועתקת לתהליך החדש, תוך סימון הדפים ב-*COW*. בעת ניסיון כתיבה יופעל חריג דף, ושגרת מערכת שתזהה חריג עקב *COW* תשכפל את המסגרת ותשנה את ההרשאות בהתאם (ותמפה אחד מהתהליכים למסגרת החדשה). בחזרה מהחריג יעשה ניסיון כתיבה נוסף שיצליח. בחלונות תהליך חדש נוצר ע"י יצירת תהליך בתול המריץ תוכנית בהנחית התהליך שקרא לו. חסכוני אם מריצים תהליך שונה, אך פחות יעיל אם מריצים את אותה תוכנית שהתהליך הקיים מריץ.

4.5 מדיניות לתזמון מעבדים:

בעת מיתוג יש קבוצת חוטים מוכנים לריצה וקבוצת חוטים שממתינים שלא מוכנים המערכת מחליטה איזה להריץ. מטרות מדיניות התזמון:

- למזער את התקורה עקב מיתוג, כלומר את הזמן הנדרש ל-*context switching*.
- ניהול זמנים לזמן תגובה נמוך ואידיאלי (אין צורך שיהיה נמוך מדי, אך מספיק נמוך – למשל תגובת עכבר).

אלו יכולים לעמוד בסתירה: מיתוג רק בזמן המתנה לאירוע יכול ליצור זמן תגובה גבוה, ומיתוג תכוף יכול ליצור תקורה גבוהה. מטרות נוספות:

- זמן תגובה מובטח לאירועים מסוימים: למשל למערכות מוטסות, מערכות זמן אמת. לינוקס ויוניקס אינן כאלה.
- תזמון דביק: במחשבים מרובי ליבות רצוי להריץ את אותו חוט/תהליך על אותו מעבד פיסי בכל ריצה, להקטנת *cahce/TLB misses*.
- תזמון כנופיות: מריבוי מעבדים רצוי להכניס את כל החוטים הלא ממתינים של תהליך אחד למעבדים שוני ם במקביל, שכן סביר שחוט אחד תלוי בפעולת חוט אחר, וכך נמנע זמן המתנה.

4.6 תזמון מעבדים עם *multilevel feedback queues*:

מנגנון מימוש מדיניות התזמון מבוסס לרוב על תורים מסוג זה , המגדירים טווח רחב של מדיניויות תזמון . ניתן לתזמן בו תהליכים /חוטים. המבנה מורכב ממערך תורים. התור במקום ה-0 (הגבוה ביותר) בעל תהליכים עם העדיפות הגבוהה ביותר וכן הלאה לשאר הרמות מכילים רק תהליכים מוכנים לריצה, ולא כאלה שממתינים . המערכת קובעת לאיזה תור יכנס כל תהליך . לכל תור פרק זמן מקס ' לריצה רצופה של תהליך בו . מוגדרים במבנה גם כללי מעבר תהליכים מתור לתור . בחירת תהליך לריצה : התהליך בראש התור הגבוה ביותר שלא ריק. מפסיק לרוץ כשמגיע לפרק הזמן המקס ' ויש תהליכים נוספים בתורו, או שתור גבוה יותר מתמלא. בחזרת התור שהופסק (אם לא הופסק מהמתנה לאירוע), יוחזר לסוף התור בו היה. חוקי העברה:

- תהליך שצבר זמן מעבד רב יועבר לתור נמוך יותר, למנוע הרעבה מתהליכים מתורים נמוכים.
- תהליך שממתין למעבד זמן רב יועבר לתור גבוה יותר כדי למנוע הרעבתו
- אירוע שתהליך מחכה לו גורם בד"כ להכנסת התהליך לתור גבוה, כיוון שלרוב האירוע מעיד שמשאב הוקצה לתהליך, ונרצה לשחררו מהר.

מערכות מאפשרות למשתמש שליטה על הפרמטרים של חוקי המעבר, ולא על העדיפויות ישירות. למשל: הגבלת גובה התור המקס' לתהליך. שתי שיטות להערכת זמן המעבד שתהליך צרך (כדי לממש מדיניות העברה): 1) לזכור האם הופסק בגלל המתנה לאירוע, מעיד שצרך זמן מעבד רב, או בגלל שנכנס תהליך גבוה יותר. 2) שערוך כל פרק זמן מסוים של זמן המעבד של כל התהליכים שקלול שערוך זה עם השערוך הקודם

פרקי זמן קצרים לתורים גבוהים וארוכים לנמוכים מממשת מדיניות זמן תגובה מהיר לתהליכים אינטראקטיביים ונצילות מעבד גבוהה שאין תהליכים אינטר': תהליכים אינטר' לרוב צורכים זמן מעבד קצר וממתינים הרבה לקלט , לכן יהיו גבוהים. תהליכים לא אינטר ' ירוצו כשאין תהליכים אינטר ' ויוכלו לרוץ פרק זמן ארוך בלי הפרעה. חסרון התורים: תלויים בפרמטרים רבים, ולכן קשים לכוונון.

פרק 5: תכנות תהליכים בו זמניים:

מספר סיבות למימוש תוכנית בריבוי חוטים בו"ז:

- ניצול מספר מעבדים פיסיים במחשב
- ניצול המעבד בזמן שתוכנית מחכה להתקן איטי כגון דיסק או רשת תקשורת
- מתן אשלית מקבילות פעולות (למשל עדכון מסך והזזת עכבר).
- מימוש שרתים בהם כל לקוח מטופל ע"י תהליך/חוט אחר.

3 האחרונות מבטאות מודולריזציה טמפורלית. שואפים לממש מודולים ללא תלות הדדית מלבד דרך הממשק המשותף שלהם , לכך משתמשים במעבד וירטואלי לכל מודול: חוט. גם המערכת מבצעת פעולות בו"ז, למשל תהליך של המערכת שאינו תלוי בתהליך משתמש שרץ במקביל

5.1 חוטים: המנגנונים:

שימוש בשלושה מנגנונים עיקריים: מנגנון יצירה, מנגנון מניעה הדדית (*mutex*) ומנגנון הודעה על אירועים. יצירה מתבצעת ע"י קריאת מערכת שמריצה שגרה ולא ממתינה לסיומה, השגרה הקוראת לא מושעית בזמן שהנקראת רצה . לכן יש צורך בתיאום, וזו מבוססת על מנעול (*lock*). מנעול יכול להכיל חוט אחד לכל היותר , וחוט יכול לקרוא לנעילת מנעול על המזהה שלו . אם המנעול תפוס, ימתין עד שיתפנה ואז ימשיך . מנעולים מבטיחים אטומיות, כלומר שסדרת פעולות של חוט רצה תוך הבטחה שחוטים אחרים לא נגישים למבנה הנתונים בעת הריצה . מנגנון תיאום נוסף הוא אירועים (*events*), שהם משתני תנאי . חוט יכול להאזין לאירוע ולהתנות את פעולתו בהתרחשות האירוע , ויכול גם לסמן שאירוע קרה . דרך זו יעילה מאשר דגימה תקופתית מבזבזת משאבים האם האירוע הנדרש קרה כבר . בעת המתנה לאירוע יש לשחרר מנעולים (אם היו) כדי לאפשר לחוטים אחרים להשתמש במשאב הנעול.

ביוניקס/לינוקס: כדי להמתין לאירוע קוראים ל-$wait(c,m)$, m הוא מנעול שצריך להיות נעול ע"י החוט הקורא. פונ' זו משחררת המנעול והחוט נכנס להמתנה. לא יתכן שבין שחרור m להמתנה לחוט אחר יקרה c. בשחרור מהמתנה נועל החוט את m שוב לפני חזרה מהפונ'. בחזרה ידוע ש-c קרה ו-m נעול. קריאה ל- $signal(c)$: פעולה המאותת ש- c קרה, משחררת את אחד החוטים הממתינים ל- c, $broadcast(c)$: משחררת את כל החוטים שממתינים ל-c. חוט שנכנס להמתנה לאחר איתות c, ימתין עד האיתות הבא (אירוע הוא הבזק , לא מצב בינארי נשמר). דוגמא: שני חוטים שאחד אחראי לכתוב לחוצץ והשני אחראי לקרוא ממנו, שימוש ב-*wait* לראשון במקרה שהחוצץ מלא (לא ניתן לכתוב עליו) ולשני כשהוא ריק. נעשה במקרה זה שימוש ב-*while* ולא ב-*if*: *while* מוודא שהתנאי כל הזמן מתקיים, *if* מוודא רק פעם אחת. כך, בחזרה מ-*wait* היינו צריכים לבדוק שוב את התנאי. בדוגמא הפשוטה הנ"ל *if* יכול להתאים, אך בתוכניות מורכבות צריך בדיקה גלובלית לנכונות ו-*while* חוסך זאת.

בדוגמא בה יש כמה חוטים קוראים וכמה כותבים , שימוש ב-*signal* יכול להוות בעיה: חוט קורא (ששחרר את החוצץ) יכול לאותת את האירוע , וזה יתפס (לא ניתן לשלוט ע"י מי) ע"י חוט קורא אחר, וכך כל החוטים (קוראים וכותבים) ישארו בהמתנה לעד. פתרון: החלפת *signal* ל-*broadcast*.

5.2 חיות וקיפאון:

שימוש לא נכון יכול להביא לקיפאון (*deadlock*) – כולם ממתינים לשחרור מנעול או אירוע תוכניות ללא קיפאון נקראות תוכניות בעלות חיות

בעיות קיפאון לרוב פשוטות לגילוי ע"י *debugger*. דוגמא לקפאון: חוט 1 נועל *m1*, חוט 2 נועל את *m2* ומנסה לנעול את *m1*, וחוט 1 מנסה לנעול את *m2*. שימוש לא נכון באירועים: דוגמא כמו קודם – שימוש ב-*signal* עם כמה קוראים/כותבים. קפאון מנעולים יותר קשה למניעה מקפאון אירועים.

מודל פורמלי: גרף דו"צ בו בצד אחד חוטים ובצד השני מנעולים. חוט נעול: קשת מהמנעול לחוט. חוט ממתין למנעול: קשת מהחוט למנעול. התוכנית בקפאון אמ"מ יש בגרף מעגל מכוון. מערכות מסדי נתונים מזהות כך קיפאון, ויכולות לשחזר את מצב הנתונים לפני פעולות החוט שהביאו לקפאון. רוב המערכות לא עושות זאת: 1) כשיש מנגנונים נוספים (אירועים, סמפורים) זה מורכב, 2) לרוב קפאון תוקע את כל החוטים ומשתמש יכול להעיף בעצמו את התוכנית, 3) ביטול פעולות שהביאו לקפאון בתוכניות רגילות(לא מעדכנות מסדי נתונים) לא אפשרית.

שתי דרכים עיקריות למנוע קפאון בגלל מנעולים /משאבים יחודיים: 1) חוט ינעל מנעול אחד בלבד בכל זמן נתון, כך לא יתכן מעגל בגרף. 2) סידור המנעולים והבטחת נעילה לפי סדר ז ה, וערבוב בדר יגרום למעגל. דרכים נוספות: מבני נתונים שהמנעולים מגינים על איבריהם, כמו למשל מנעול לכל איבר ברשימה מקושרת, מסודר לפי סדר הרשימה. שיטה זו לא תמיד עובדת, למשל עץ שסורקים אותו גם מהשורש וגם מהעלים (סדר לא טוב פה). דרך טובה היא הגבלת זמן ההמתנה לאירוע, תלוי ביכולת החוט להתקדם גם אם האירוע לא ארע

5.3 גרעיניות:

נעילות ואירועים כופים סדר חלקי של ביצוע פעולות ע"י חוטים שונים. שימוש ביותר מנעולים ואירועים בד"כ מורשים יותר סדרים חלקיים, כלומר פחות אילוצים על הסדר, לכן פחות המתנה וביצוע מהיר יותר. דוגמא: רשימה מקושרת עם מנעול על שימוש על כל הרשימה שיפור: לאפשר לשני חוטים לעבוד על הרשימה בו"ז, בתנאי שלא על שני איברים סמוכים (יותר מנעולים, ייעול). דרך נוספת: מנעולים על כל איברי הרשימה, ולמשל נעילת שלושה איברים כאשר רוצים למחוק את האמצעי מביניהם. מאפשר יותר בו-זמניות. מעט מנעולים = גרעיניות (*granularity*) גסה. ריבוי מנעולים והגדלת בו-זמניות: גרעיניות עדינה (*fine* לעומת *coarse*). לגרעיניות עדינה מחיר של צורך לנעול ולשחרר יותר מנעולים בדרך, ובכך לצרוך משאבים. גרעיניות אופטימלית תלויה במחיר פעולות מנעולים, מספר מעבדים, משך זמן טיפול חוט במבנה נתונים, מספר החוטים שיפיקו תועלת מבו"ז ועוד.

5.4 יעילות:

מעבר בין חוטים מבזבז זמן מעבד, ולכן רוצים למנוע מיתוג. מיתוג חוטים זול ממיתוג תהליכים (מרחבי זכרון שונים), אך עדיין יקר. מיתוג מיותר בין חוטים קורה: 1) שחרור חוט שלא מסוגל להתקדם מהמתנה, מגלה שלא יכול להתקדם וחוזר להמתנה. למשל, בדוגמת קריאה/כתיבה לחוצץ – כל החוטים מתעוררים למשל מאירוע שכותב שחרר, אך כל הכותבים האחרים התעוררו לשווא. פתרון: הפרדת אירועים לאירוע חוצץ ריק ואירוע חוצץ מלא (חוסך התעוררות). כך אפשר גם להשתמש רק ב-*signal* ובכלל לחסוך התעוררות מרובה (אירועים שונים מבטיחים חיות). 2) חוטים רבים מתחרים על מעבדים מעטים. המערכת יכולה למתג בין חוטים לא ממתינים רבים, והתקורה גבוהה. רצוי שמספר החוטים הלא ממתינים יהיה כמספר המעבדים.

5.5 הגינות והרעבה:

המערכת לא מבטיחה הגינות במובן שחוטים שמבקשים לנעול מנעול לא בהכרח ינעלו אותו לפי סדר בקשתם. סיבות: 1) יתכן ששתי בקשות התקבלו באותו זמן ממש על שני מעבדים שונים, ואז אין ביניהם סדר. 2) מנגנון הנעילה ממומש לרוב ע"י חומרה, שלא מבטיחה הגינות. המערכת כן מבטיחה מניעת הרעבה. גם אם המערכת היתה מבטיחה הגינות, תוכניות לא היו מתנהגות בהכרח בהגינות, בשל מורכבות המבנה שלה למשל.

5.6 עדיפויות והיפוך עדיפויות:

במערכות מסוימות נ יתן להגדיר עדיפויות לחוטים: יותר זמן מעבד, קדימות בקבלת מנעול ואירוע. מימוש פשטני של מנגנון זה עלול להביא להיפוך עדיפויות (*priority inversion*): חוט בינוני (מבחינת עדיפות) מונע מחוט נמוך לרוץ, וזה תופס משאב לחוט גבוה. כדי למנוע זאת יש להעלות זמנית חוט המחזיק מנעול לעדיפות של החוט בעדיפות הגבוהה ביותר שמחכה למנעול זה לא ניתן למימוש על אירועים כי לא ידוע האם ומתי חוט יודיע על אירוע

5.7 נושאים נוספים ב-*Posix threads*:

ביוניקס ולינוקס מנעולים ואירועים נגישים רק לתהליך בודד וניתן לממש אותם ביעילות. שירותי *posix* נוספים: מנעולי קריאה/כתיבה, המתנה לסיום ריצת חוט (*join*) והפסקת ריצת חוט (*cancel*), משתנים לא משותפים (*keys*), שליטה בעדיפויות ובמדיניות תזמון מעבדים.

5.8 חוטים ב-*win32*:

ב-*posix* אירוע הוא מאורע רגעי שמשחרר רק חוטים שממתינים באותו רגע לאירוע. בחלונות אירוע הוא מצב בינארי סטטי שנשמר שינויו, וביציאה מהמתנה המנעול לא משתחרר. אירועים מתחלקים בחלונות לאיפוס ידני ואיפוס אוטומטי. ידני: קריאה ידני לקריאת מערכת מדליקה /מכבה את האירוע. הדלקה משחררת את כל החוטים שממתינים. חוטים שנכנסים להמתנה כשהאירוע דולק לא ימתינו. אוטומטי: הדלקה ידנית אך כבה אוטו' כאשר חוט משתחרר מהמתנה לאירוע. אם אין אירוע ממתין, ישאר דולק עד שיגיע אחד כזה. הבדל נוסף מ-*posix*: ניתן לתת להם שמות ולגשת אליהם מתהליכים אחרים. קטע קריטי (*critical section*): ניתן לשימוש ע"י חוטים באותו תהליך, תקורה נמוכה יותר (*Mutex* פרטי לתהליך).

פרק 6: מערכות קבצים:

המונח מתייחס לשני מושגים:

- צורת הארגון של הקבצים (רצפי נתונים) בדיסק ולשגרות שמטפלות במבנה הנתונים. רוב המערכות תומכות בכמה (חלונות ב-*NTFS*, *FAT*).

- התייחסות למבנה נתונים ספציפי של קבצים בד"כ על דיסק או מחיצה, ולמרחב השמות המתייחסים לקבצים אלו.

קובץ (*file*): רצף נתונים, מערך בתים שגודלו לא קבוע מראש. קריאה וכתיבה ניתנת מכל מקום ברצף, חריגה מחזירה לרוב שגיאה. יתכנו חורים ברצף שיתמלאו בזבל או שקריאה מהם תחזיר שגיאה, תלוי מערכת. מערכות מסויימות מתייחסות לקובץ כעצם מורכב של מספר רצפים. למשל חלונות תומכת במספר לא מוגבל של רצפים לקובץ. תתי הרצפים הן עידון מרחב השמות של מערכת הקבצים. ניתן לדמותו למדריך (*directory*) בעל מספר קבצים. ההבדל הוא שפעולות על הקובץ משפיעות אוטומטית על כל תתי הרצפים שלו (מחיקה, שינוי הרשאות למשל). יתרון נוסף הוא יכולת עדכון תתי רצף ע"י תוכנית אחת ללא ידיעת תוכניות אחרות (למשל תוכנת גיבוי המוסיפה תאריך גיבוי אחרון לקובץ). חסרון: לא תמיד נתמך ע"י מערכות אחרות או אפילו ע"י מערכות קבצים אחרות שנתמכות ע"י אותה מערכת (הפעלה). מעתה נתייחס לקובץ כבעל רצף אחד. סוגי שירותים עיקריים של מערכת ק':

- מרחב שמות קריא לבני אדם.
- מנגנון לשמירת ואחזור קבצים על התקני זכרון חיצוניים כגון דיסקים מגנטיים ואופטיים

6.1 מרחב השמות:

מערכות תומכות במרחב שמות היררכי מבוסס מחרוזות להתייחסות לעצמים. מרחב השמות: גרף מכוון ללא מעגלים עם שורש. בחלונות מציינים רצף צמתים בגרף, עצמים, בלינוקס/יוניקס אין לעצמים שמות אלא מציינים שרשור קשתות בגרף. דהיינו מצביעים. 4 סוגי עצמים ב-*/home/a.out*:

- מדריך, כגון / או *home/*. עצמים שמכילים קשתות במרחב השמות, מצביעים לעצמים אחרים.
- קבצים רגילים שמכילים נתונים, כמו: */home/a.out*. תמיד עלים במרחב השמות.
- התייחסות להתקני חומרה, כמו */dev/hda*. גם עלים במרחב השמות. כאן: מציין את ההתקן ה-0 שמנוהל ע"י מנהל התקן 6 בטבלת עם הטמנה.
- מצביעים סימובליים (*symbolic links*), כגון: */lib/xyz*.

במרחב השמות יכולים להופיע עוד סוגי עצמים, כגון ערוצי תקשורת, צינורות ושקעים. חלונות תומכת גם במנעולים ואירועים במרחב השמות. בחלונות מרחב השמות מסובך יותר כי: 1) לעצמים עצמם יש שמות, ויתכנו שמות נרדפים לאותו עצם, למשל שם תואם *MSDOS* (11 תוים). 2) מרחב השמות מורכב מכמה מדריכים ברמה העליונה, ולכל אחד תת מרחב לפענוח שמות עצמים מסוג מסוים. למשל: שמות קבצים מפוענחים כאילו מתחילים ב-*??* ושמות עצמים נדיפים (מנעולים) כאילו ב-*BaseNameObjects*. ממשק המערכת מאפשר לתוכניות ביצוע פעולות על מרחב השמות. למשל בלינוקס/יוניקס (דומה בחלונות):

- מחיקת מצביע: *unlink*. אם אין מסלול מהשורש לעצם, גם העצם נמחק.
- שתילת מערכת קבצים שלמה במקום מסוים במרחב הקיים: *mount*. הסרה: *unmount*.
- פענוח שם למזהה פנימי והודעה למערכת שבשימוש שלא ימחק: *open*.
- הודעה על סיום שימוש בעצם: *close*.
- יצירת עצם חדש ומצביע אליו ממדריך כלשהו: *open, create* או *mknod* וכו'.
- יצירת מדריך חדש ומצביע אליו: *mkdir*.
- יצירת מצביע לעצם קיים: *link*.
- יצירת מצביע סימבולי: *symlink*.
- שינוי שם מצביע: *rename*.

6.1.1. פענוח שמות, מצביעים סימבוליים ונקודות הצבה:

קריאת *open* מקבלת מסלול בגרף ומחזירה מזהה שיהיה ניתן להתייחס לעצם בהמשך. פענוח */home/a.out* למשל: חיפוש המצביע *home* בספריית השורש, שם חיפוש מצביע בשם *a.out* שמחזיר מזהה לעצם שהמצביע מוביל אליו. כל מדריך מכיל שני מצביעים קבועים: . מצביע לעצמו, .. מצביע למדריך ההורה. לכל תהליך המערכת מחזיקה מצביע למדריך הנוכחי. שמות שלא מתחילים בתו הפרדה מפוענחים החל מהמדריך הנוכחי (*path* יחסי). מסלולים שמתחילים בו נקראים מסלולים יחסיים, בניגוד למוחלטים שמתחילים בשורש. הפענוח דורש משאבים, חיפוש בכל מדריך. אם מדריך לא בזכרון יש להביאו מהדיסק (או חלק ממנו). החיפוש לוקח זמן מעבד, ושמות קצרים מהירים לפענוח מארוכים. בפרט, פענוח יחסיים מהיר ממוחלטים.
מצביעים סימבוליים: קבצים מיוחדים המכילים שם קובץ. אם בפענוח מתגלה שתחילית מסוימת היא מצביע סימבולי, הפענוח מתחיל מחדש עם החלפת התחילית לתוכן המצביע הסימבולי. המערכת לא מוודאת ש עצם קיים כשיוצרים אליו מצביע סימבולי, ולא מתחשבת בכאלו במחיקת עצם. מצביע סימבולי מאט את קצב הפענוח.

נקודות הצבה/פענוח (*mount/reparse points*): ביוניקס נשמרת טבלת צמתים ממרחב השמות המצביעים על שורשי מערכות קבצים שלמות. בהגעת פענוח לנקודה זו, הפענוח ממשיך במערכת הקבצים שמוצבת בה. נקודת ההצבה חייבת להיות מדריך קיים. בחלונות: נקודות פענוח הם צמתים במערכת הקבצים שמפנים את הפענוח לשורש של מערכת קבצים אחרת, מדריך בכזו או שגרת פענוח כלשהי. הראשון כמו ביוניקס, השני קצת יותר עדין. השלישי: ניתן לממש מערכות אחסון היררכיות: מעבירות קבצים שלא ניגשו אליהם זמן רב לאחסון במדיה איטית. בעת פענוח, תופעל מערכת האחסון

ותשחזר את הקובץ למערכת הקבצים . בשיטות אלו נתפר מרחב שמות אחיד . ביוניקס הפענוח נשמר במבנה נדיף של המערכת ובחלונות נשמרות נק ' הפענוח בתוך מע' הקבצים. ביוניקס צריך להציב מחדש בכל אתחול מערכת, בחלונות לא. יתרון: ניתן להעתיק מע' קבצים בלי נק' פענוח ביתר קלות.

6.1.2. מעגלים ומחיקת קבצים:

גרף מרחב השמות אינו עץ ולכן יתכנו בו מעגלים, מקשה על הבחנה בין קובץ נגיש שאין למחקו ובין קובץ ללא מסלול (*garbage*). זיהוי זבל ע"י ספירת הצבעות. בהגעה ל-0, נמחק הקובץ, לא יעיל אם יש מעגלים. לכן המערכת לא מתירה ליצור מעגלים , למשל ע"י איסור יותר ממצביע אחד למדריך . גם אם יש מעגלים, ניתן לזהות זבל ב-*garbage collecting*: סימון קבצים נגישים במרחב השמות ומחיקת כל השאר. בד"כ לא בשימוש מערכות הפעלה. קיום מצביע סימבולי לקובץ לא מונע את מחיקתו, ולכן מערכות מתירות יצירת מעגלים ע"י מצביעים סימבוליים. קבצים פתוחים לא נמחקים. אם מס' הצבעות לקובץ יורד ל-0, ימחק ררק כאשר לא יהיו עוד תהליכים בהם הוא פתוח

<u>6.2 שמירת ואחזור קבצים</u>:

6.2.1. סמנטיקה של גישה לקבצים:

סמנטיקת גישה לקבצים היא חוקי ההתנהגות של ג ישת תהליכים רבים לאותו קובץ (שיתוף מידע ביניהם). <u>קונסיסטנטיות</u>: סמנטיקה פשוטה עבור גישות לקבצים ששמורים על דיסק מקומי: 1) כתיבות הן אטומיות: אם רצף נכתב וחלקו נקרא, הקורא יראה או את המצב לפני או אחרי הכתיבה , אך לא מוערבב. 2) כתיבה לקובץ נראית מיד לתהליכים א חרים, כלומר קונסיסטנטיות סדרתית. גישות לגבצים לא על אותו דיסק לרוב לא מקיימות זאת . <u>עמידות</u>: אירוע נפילת מחשב יכולה לגרום לביטול פעולות שלכאורה הסתיימו. למשל, כתיבת תהליך לדיסק ונפילת המחשב: התהליך מבחינתו כתב אך המידע היה בחוצץ ולא הועבר לדיסק בנפילה , ואבד. פעולה זו אינה עמידה. ניתן לדרוש עמידות מפעולות כתיבה, יצירה או ביטול קובץ. רוב המערכות לא מבטיחות עמידות לכתיבה , אך כן לעמידות נתונים לאחר *close*. חלק מבטיחות עמידות לשינויים במרחב השמות . לרוב יש פגיעה בביצועים בגלל שהמערכת מכריחה כתיבה מיידית לדיסק (לפי חזרת קריאת המערכת), במקום דחיית כתיבות לממזעור המתנה לתזוזת ראש הדיסק.

6.2.2. מיפוי קבצים:

קבצים שמורים בבלוקים בגודל קבוע ואינם בהכרח רצופים בדיסק ניסיון הקצאת בלוקים רצופים: 1) בעיה להגדיל את הקובץ, מצריך להעתיקו מחדש במקום אחר. 2) פרגמטציה חיצונית, כלומר הווצרות חורים בין קבצים שיהיו לא שמישים. בלוקים גדולים משפרים זמן גישה ממוצע כי ממתינים פחות לתנועת הזרוע וסיבוב הדיסק. מקטין גם את מבנה הנתונים שממפה קבצים לבלוקים. אבל, בלוקים גדולים מבזבזים מקום, כי לרוב האחרון לא מלא. מערכות קבצים יכולות להשתמש בטווח גדלים בתנאי שכל הקבצים בעלי אותו גודל בלוק , בד"כ 1-8 ק"ב. לרוב המערכת תבחר בלוקים גדולים לדיסקים גודלים (ליעול) וקטנים לקטנים (למנוע בזבוז מקום). יש מערכות קבצים שמאפשרות להחזיק בלוק "זנבות" קבצים, שכל זנב שמור על שבר (*fragment*) של בלוק. לא משפיע על זמן הגישה (רוב המידע על בלוקים שלמים) אך מסבך את מבנה הנתונים הממפה את הנתונים לבלוקים בדיסק.

<u>*Inode*</u>: בלינוקס/יוניקס. קובץ מיוצג ע"י *inode* שמחזיק, יחד עם הרשאות , זמן יצירה , אורך ומערך אינדקסים של בלוקים בדיסק (מיפוי). מערך האינדקסים ב-*inode* מוגבל בגודל, ולקבצים גדולים משתמשים במיפוי היררכי. ב-*inode* מערך לא גדול של מצביעים לבלוקים , אחריהם מצביע נוסף לבלוק המכונה *indirect block* שמחזיק אינדקסים של בלוקי נתונים בהמשך הרצף . אחריו בלוק *double indirect* המצביע למערך *indirect*, לבסוף *triple indirect* באותו הגיון. גם כאן מוגבל, אך גדול. אם כל בלוק מכיל d מצביעים, ול-n *inode* מצביעים ישירים, סה"כ: $n + d + d^2 + d^3$ בלוקים. גישה לקובץ גדול תדרוש 5 גישות (*inode*, 3 בלוקי מצביעים, בלוק נתונים) ולקובץ קטן רק 2. ה-*inode* נשמר בזכרון לקבצים פתוחים. לרוב *inodes* נשמרים בנפרד מנתונים באזור מסוים בדיסק. כך בהבאת *inode* מובאים נוספים (האצת פתיחת קבצים וסריקת מדריכים), מצד שני יתכן בזבוז מקום - לא מספיק ל-*inode* אך כן לנתונים, או אין מקום לנתונים פרט באזור זה. מערכות מתקדמות מקצות מקום ל-*inode* לפי צורך.

<u>*NTFS*</u>: בחלונות. רשומות הם מקבילים במערכת זו ל-*inodes*. קובץ מיוצג ע"י רשומה, כולן שמורות במערך בקובץ *MFT* (*master file table*). סהו קובץ רגיל, ולכן משתנה גודלו לפי צורך (לא צריך להקצות מראש מקום). רשומות גדולות מ-*inodes* ויכולות להכיל את תחילת הקובץ עצמו, כך קבצים ומדריכים קטנים נשמרים ברשומה - זמן גישה קצר.

<u>*FAT*</u>: *file allocation system*, משמשת *DOS* ו-95/98. טבלה אחת של מצביעים משמשת למיפוי כל קבצי המערכת . שמורה בתחילת דיסק /מחיצה ומכילה כניסה לכל בלוק במערכת הקבצים , קובץ מיוצג ע"י מס' הבלוק הראשון שלו. בלוקים של קובץ שמורים כרשימה מקשורת בטבלת המצביעים. בלוקים לא מוקצים שמורים ברשימה נוספת חסרון: זמן גישה ארוך לקבצים גדולים (סריקת כל רשימת הבלוקים עד הרצוי) וקושי הקצאת בלוקים.

6.2.3. מדיניות להקצאת בלוקים לקבצים:

מדיניות זו משפיעה על ביצועי מערכת הקבצים וצריכה לשאוף ל:

- הקצאת רצפים פיסיים ארוכים של בלוקים לרצפים ארוכים בקובץ, או הקצאות בלוקים קרובים אחרת. על שום עקרון רציפות במקום.
- הקצאת קבצים שמשתמשים בהם יחד קרוב על הדיסק. בפרט קבצים מאותו מדריך וקרובים בזמן כתיבתם.

בעבר בלוקים חופשיים נשמרו ברשימה מקושרת בדיסק . הקצאה מתחילה מתחילת רשימה זו כדי לחסוך זמן. זה יוצר סדר שרירותי של בלוקים במערכת הקבצים (מחיקת קבצים מחזירה בלוקים חופשיים לרשימה). קבצים יהיו מפוזרים על גבי הדיסק, מעלה את קצב ההעברה מהזכרון לדיסק. בשיטה זו ניתן לשפר ביצועים ע"י *defragmentation* מדי פעם, כלומר בניה מחדש של הקבצים על הדיסק. תהליך זה יקר ואיטי ככל שהדיסק מלא.

<u>*FFS*</u>: *fast file system*: מערכת קבצים המקצה בלוקים בצורה מתוחכמת, מחלקת את הדיסק ל קבוצות גלילים רצופים, כדי להקצות נתונים קשורים תוך מזעור זמן המתנה לתזוזת הראש . בכל קבוצה יש *superblock* (מבנה נתונים למערכת הקבצים הספציפית), מאגר *inodes*, מערך ביטים לסימון בלוקים פנויים ועוד. מס' ה-*inode*: 1 לכל *2kb* נתונים (אידיאלי למניעת בזבוז מקום). מדיניות דו שלבית להקצאת בלוקים לקובץ חדש/הגדלת קיים:

- בחירת קב' גלילים בה יוקצו הבלוקים. אם הקובץ קיים, יש מקום בקבוצה בה מוקצה, ולא תופס יותר מ- 25% ממנה, יוקצה לו מקום נוסף שם. אחרת יוקצה לו מקום לגדול בקבוצה אחרת בה הרבה מקום, תוך הגבלת הגדילה (לשמור מקום לגדילת אחרים בקבוצה). הקבוצה המועדפת היא זו של המדריך שבו הקובץ מוקצה.
- הבלוק שיוקצה יוקצה למקום הקרוב ביותר מבחינה סיבובית לבלוק האחרון הנוכחי של הקובץ. אם לא פנוי, יבחר ע"י מדיניות משנית (לא חשוב).
- *FFS* משתדלת לעבוד עם רצפים רצופים של *64kb* כדי לאפשר למנהל ההתקן לבקש העברה גדולה של נתונים מהבקר. אלו נקראים *clusters*.

תוצאות מדיניות ההקצאה:

- קבצים תחת אותו מדריך (קשורים) נשמרים קרוב מבחינת נתונים וה-*inodes* שלהם (לפחות תחילת רצף הנתונים).
- בלוקים של קובץ מוקצים בד"כ ברצפים של כמה *MB* ששמורים ברצף סיבובי אופטימלי.
- גישה יעילה לבלוקים של *64kb* או יותר כיוון שאלו נשמרים ברצף על הדיסק(*clusters*).

מע' *FFS* מעבירה נתונים בקצב של 50-100% מקצב ההעברה המקס' של הדיסק. הביצועים עולים ככל שהגישה לבלוקים גודלים יותר(גישה ל-*cluster*). ביצועי מערכת זו נמוכים כשכותבים לכמה קבצים, ויש תזוזות רבות של הראש, כמו כן ביצירה ומחיקה תכופה.

<u>*LFS*</u>: מערכת הקבצים היומנאית. התייחסות לדיסק כסרט אינסופי שנתונים תמיד נכתבים לסופו (חדשים או שכתוב בלוקים קיימים). כך, בעת כתיבות הראש זז ברציפות. בקריאות הראש זז לפי בלוקים, אך מניחים שנתונים שנכתבים יחד יקראו יחד, ולכן רצופים. בשכתוב בלוק, העותק החדש יכתב בסוף. כל הבלוקים שמובילים לבלוק הנתונים, ה-*inode* ובלוקי מצביעים שמצריכים שינוי גם ישוכתבו לסוף כיוון שה-*inode* משוכתב כל שינוי בבלוק, מתייחסים לקובץ ע"י מזהה *inode* לוגי. קובץ מיוחד מחזיק מיפוי מספרי *inodes* למיקום פיסי, זהו קובץ רגיל במערכת הקבצים. בכל כתיבת בלוק נכתב לאיזה קובץ שייך ומיקומו ברצף נתוני ה קובץ. לרוב כתיבה לא נעשית ישירות לדיסק, אלא מצטברת בחוצץ (או מטמון), ואז בכתיבה נעשה מיון של הבלוקים לפי קובץ ומיקום בקובץ.

מערכת *LFS* מחלקת הדיסק ל-*segments* בגודל מליון בתים (~), הסרט הוא רשימה מקושרת שלהם. החלק הכתוב של הרשימה מכיל בלוקים חיים ובלוקי זבל שכבר שוכתבו. על המערכת לדאוג תמיד בסוף למקטעים נקיים, ולכן יש *cleaner* שרץ ברקע ומנקה מקטעים (סגמנטים) ע"י הפרדה בין בלוקים חיים לזבל ע"י שכתוב החיים לסוף הרשימה. אז מסמן את המקטע שנוקה כריק ומעבירו לסוף הרשימה. בדיקת חיות: ה-*cleaner* קורא מהבלוק את הקובץ ומיקומו בקובץ, ומבקשת מיפוי של נתון זה ממערכת הקבצים. אם מקבל תשובה שונה, אזי בלוק זה זבל, אחרת הוא חי. לביצועים טובים יש לבחור מקטעים בעלי מספר קטן של בלוקים חיים, וה-*LFS* מחזיקה מבנה נתונים עזר ל-*cleaner*.

שתי המערכות משתמשות במידע עבר לחיזוי נתונים שיקראו בעתיד. ה-*LFS* בקרבה טמפורלית: נתונים שנכתבו יחד יקראו יחד. ה-*FFS* בקרבה לוגית במרחב השמות: נתונים קרובים באותו קובץ או באותו מדריך יקראו יחד.

6.2.4. נפילות והתאוששות:

מצבים לא קונסיסטנטיים של מערכת קבצים כוללים: בלוק פיסי המסומן פנוי אך ממופה כחלק מקובץ, בלוק פיסי ממופה לשני קבצים או יותר, בלוק פיסי לא ממופה אך לא מסומן פנוי, מצביע במדריך ל-*inode* המסומן פנוי, קובץ (*inode*) ללא מצביעים אך לא מסומן פנוי. מעבר ממצב קונ' אחד לאחר עוברת דרך מצבים לא קונ'. למשל מחיקה דורשת: מחיקת מצביע מהמדריך, סימון *inode* כפנוי, סימון הבלוקים כפנויים. מצבי ביניים כאן לא קונ'. שגרות של מע' הקבצים פועלות נכון רק אם מבנה הנתונים קונ'. אם הכל תקין, ניתן להבטיח קונ' לכל שגרה ע"י שימוש במנעולים. אם המחשב נופל במצב ביניים, מע' הקבצים אינה קונ', ונדרש *recovery* כדי לאפשר פעילות תקינה. כדי לוודא קונ', נשמר בדיסק ביט המייצג קונ'. בפעילות מסודרת הכתיבה האחרונה מדליקה ביט זה. בתחילת פעולת מע' הקבצים, תחילה מכבה ביט זה. ישנן מספר גישות להחזרת קונ', להלן.

<u>מערכות קבצים עם כתיבה זהירה</u> (*careful-write*): כתיבות מתבצעות כך שכל מצבי הביניים הלא קונ' יהיו כך שבהם מע' הקבצים יכולה להמשיך לפעול. בנוסף, משנות את מבנה הנתונים בדיסק מיידית. הבעיות שעלולות כן להווצר: אובדן משאבים עקב בלוקים/*inodes* לא בשימוש אך לא מסומנים פנויים. מערכות כאלה: *FAT*, *FFS*, *LFS*. סדר מחיקה: מחיקת מצביע מהמדריך, סימון *inode* כפנוי (מונע שיוך בלוק לשני *inodes*), ולבסוף סימון הבלוקים כפנויים. בחזרת מערכת לפעילות במצב לא קונ', ניתן ישר לחזור לפעול, ובמקביל תיקון רץ ברקע (בעיקר סימון *inodes* ובלוקים לא בשימוש

כפנויים). כתיבות נעשות מיידית, כלומר מערכת הקבצים נעולה לתהליכים אחרים, כדי להבטיח שסדר השינוי יעשה כנדרש. זאת הגבלה של המערכת, שכן מונע תזמון כתיבות לדיסק באופן יעיל, בעיקר בסביבות מחיקה ויצירת קבצים מרובה. בסביבות בהן רוב הגישות הן לקריאות, ביצועים טובים.

מערכות קבצים עם עדכונים רכים (*soft updates*): בכל שינוי מבני הנתונים בדיסק, מע' הקבצים מוסיפה את העדכונים הנדרשים למבנה נתונים בזכרון. לכל עדכון נשמר זהות עדכונים שיש לבצע לפני שמבצעים אותו כדי למנוע בעייתיות. כאשר מוחלט כתיבת בלוק לדיסק, המע' סורקת מבנה נתונים זה ומבצעת את כל העדכונים הנדרשים לפי הסדר לעותק של הבלוק בזכרון ואז כותבת אותו לדיסק. עותק זה נשמר בזכרון לעדכונים בהמשך.

מערכות קבצים עם כתיבה עצלה (*lazy-write*): כתיבה מהזכרון לדיסק בעיקר משיקולי ביצועים, פחות שיקולי קונ'. במקרה כזה המע' יכולה להיות לא קונ' לאחר נפילה, ונדרשת הרצת תהליך תיקון. ביצועים טובים אך *recovery* ארוך, ובמע' גדולה עשוי לארוך שעות.

מערכות קבצים מתאוששות (*recoverable*): ביצועים טובים ועליה מהירה לאחר קריסה. שימוש בטכנולוגיה של מסדי נתונים להגן על קונ' מבנה הנתונים של מע' הקבצים, אך לא מגנה באופן דומה על קבצי משתמשים. כיוון שהחזרה למצב קונ' מהירה, ניתן להגדיל את הזמן בין כתיבות בלוקים מלוכלכים לדיסק – שיפור ביצועים ע"ח סיכון גבוה לקבצי משתמשים. דוגמא: *NTFS*: כל שינוי במבנה הנתונים (להבדיל משינוי נתון בקובץ) נקרא תנועה. *NTFS* מבטיחה תנועות אטומיות, גם בנפילה: או שכולן יתבצעו או שאף אחת לא. בזכרון שני מבני נתונים לטבלת התנועות וטבלת הבלוקים המלוכלכים וקובץ *log* בדיסק. בטבלת התנועות: תנועות שלא התבצעו. ה-*log* שומר עותקים ישנים לשתי הטבלאות, רשומות לתנועות ולסיום תנועות. כתיבה ליומן ושינוי מבנה הנתונים מתבצע לעותק בזכרון (לא לדיסק). כתיבה לדיסק משיקולי ביצועים בעיקר. *NTFS* יכול לכפות אילוצי סדר כתיבה לדיסק. לביצוע פעולה נוצרת תנועה חדשה שנכנסת לטבלת התנועות כל פעולה בתנועה נעשית על הזכרון ונכתבת כרשומה משורשרת לשאר ביומן. בסוף נרשמת רשומת סיום. רשומה תכתב קודם ליומן בדיסק לפני שעשייתה תתעדכן בדיסק, וכתיבת הרשומות לפי סדר ביצוע. רשומה כוללת מידע ל-*redo* ול-*undo* בצורה אידמפוטנטית, כלומר ניתן לחזור עליה כמה פעמים ולקבל אותה תוצאה (כמו: "כתוב 1" לעומת "שנה ערך בוליאני"), זאת כיוון שלא ידוע אם בלוק הספיק להכתב לדיסק. לאחר נפילה יש ניתוח של היומן. מטרותיו: 1) זיהוי תנועות שיש להן זכר ביומן שלא נעשו סופית, להחליט איזה לסיים ואיזה לבטל: תנועות שרשומת סיומן ביומן בדיסק יבוצעו, ואלו שלא יבוטלו. לא ניתן לבצע את האחרונות כי לא ידוע אם רשימת הפעולות המלאה שלהן ביומן (יתכנו עוד שהיו בזכרון ואבדו). 2) ביצוע וביטול לפי ההחלטה קודם. אלג' ניתוח היומן מורכב.

שימוש בזכרון לא נדיף: מערכות מסוימות משתמשות בזכרון לא נדיף, שאמור להיות עמיד בפני נפילות ומהיר (שישמש כזכרון פיסי). יכול להיות זכרון רגיל מגובה בסוללה/אל פסק או זכרון מגנטי מהיר. ניתן לשמור בו מידע לשמירת קונ' מע' הקבצים. שיפור ביצועים: ויתור על כתיבה תקופתית של בלוקים לדיסק, שמירת היומן בזכרון זה למערכות מתאוששות.

פרק 7: רשתות תקשורת ופרוטוקול האינטרנט *IP*:

תוכניות הנדרשות לתקשורת מחשבים צריכות קשר בין שני תהליכים על שני מחשבים שונים. קשר צריך להיות אמין וסדור. יכולת חומרת תקשורת נמוכה מדרישות תוכניות, החומרה לא אמינה ולא סדירה. מערכת ההפעלה צריכה להתגבר על פערים אלו. תקשורת בין מחשבים או תהליכים נעשית ע"י פרוטוקולים המגדירים מבנה הודעה ומנגנון כיתוב (*addressing*). חלק מכילים מנגנוני גילוי ותיקון שגיאות ומניעת עומס. אלו מוגדרים בשכבות, שהנמוכה מגדירה שירותים בסיסיים, וכל אחת מעליה מתבססת על זו מתחתיה. חלוקת השכבות הטבעית הנפוצה:

- שכבה פיסית (*physical*): הגדרת מאפיינים פיסיקליים של העברת אות בתווך. בתקשורת קוית למשל יוגדרו מספר החוטים ותפקידם, מתחים וזרמים. באלחוטית: תדרים וכו'. פרוטוקולים ממומשים ע"י בקר התקשורת (התקן חומרה) ולא ע"י המערכת.
- שכבת המיקשר (*link*): מגדירה פרמטרים להעברת חבילות מידע בדידות בין מחשבים - מנות (*packets*). רוב הפרוטוקולים בשכבה זו מגבילים את גודל המנה. יכולים להיות לא אמינים (מנה תשלח אך לא תגיע). כוללים לרוב תיקון שגיאות, בסיס לשכבות מעליהן. פרוטוקולים ממומשים ע"י מנהל ההתקן של בקר התקשורת בסיוע מנגנוני חומרה בבקר.
- שכבת הרשת (*network*): מגדירה פרוטוקולים לנתוב מנות בין מחשבים גם אם לא מחוברים פיסית, אלא יש לנתבם דרך מחשבים אחרים, כמו פרוטוקול האינטרנט *IP*, שאינו אמין או סדור גם אם שכבת המיקשר כן.
- שכבת ההעברה (*transport*): פרוטוקולים לשליחת הודעות לתהליך מסוים במחשב אחר. חלק מוסיפים שירותי קשר סדור ואמין. שניים עיקריים: *UDP*, כמו *IP* רק בין תהליכים ולא בין מחשבים, לא מבטיח הגעת מנות ליעדן. ה-*TCP*: כן מאפשר קשר אמין וסדור בין תהליכים, ממומש ע"ב *IP*.
- שכבת היישום (*application*): מגדירה פרוטוקולים בין יישומים כמו *HTTP* להעברת דפי אינטרנט. לרוב ללא נגיעת המערכת.

פרוטוקולים מוגדרים לרוב במשפחות, כמו פרוטוקולי אינטרנט: *IP*, *TCP*, *UDP*, *HTTP*, *FTP* ועוד. רכיבי התוכנה במערכת שמטפלים בפרוטוקולי תקשורת קוריים מחסנית (*stack*) בגלל אופן הטיפול השכבתי בעת קבלת מנה, וטיפול בסדר הפוך בעת הוצאתה.

7.1 השכבה הפיסית ושכבת המקשר:

רשתות תקשורת מקומיות (*LAN*): מחברות קבוצת מחשבים קטנה במיקום גיאוג' קרוב. נפוץ: פרוטוקול ה-*Ethernet* הכולל הגדרות לשכבה הפיסית והמקשר. פעלו בקצב *10Mb/s* וחיברו מחשבים על מקטע (*segment*): כל שידור התקבל ע"י כל המחשבים המחוברים אליו, ויכול להתמען לאחד או

לכולם. רק אחד יכול לשדר בכל זמן נתון . אם יש הרבה על מקטע, הערוץ עלול להסתם. לכן עברו למבנה ממותג בו כולם מחוברים ל-*switch* שאחראי על העברות המנות ברשת. כך כל המחשבים יכולים לשדר/לקלוט במקסימום המתאפשר. מערכות מחשבים לא מבחינות בין מקטע רגיל לממותג גם קצב התקשורת השתפר עד 100 וגם 1000 מגהביט לשניה (10/100 נפוץ). בעבר התבססו על חיווט קואקסיאלי רחב, וכיום על *cat5* (כמו חוטי טלפון).

אחרות: *token ring* פותחה ע"י *IBM* (אסימון עובר במעגל ומי שמחזיק בו יכול לדבר, ניצול לא אופטימלי), התחרתה עם האתרנט אך השימוש בה דעך, כמו גם ב-*decnet* של *digital*. רשתות מקומיות אלחוטיות: מתג אלחוטי ובקרים אלחוטיים במחשבים, איטית יותר מקוית אך זולה יחסית לאבזור.

מודמים וקוי טלפון: מודם הוא התקן תקשורת טורי המחבר בין בקר מחשב לקו טלפון אנלוגי . מודם יוצר תקשורת עם מודם אחר ופקודות מועברות ישירות למודם בצד השני אל המחשב אליו מחובר . המרת מידע דיגיטלי לקול להעברה בתווך זה . מהירות התקשורת מוגבלת בגלל שתקשורת מרכזיות הטלפון היא ספרתית ומוגבלת מאוד. *ISDN*: גם רץ בתווך קוי טלפון על שני ערוצים לדיבור ולנתונים שניתנים לאיחוד להעברה מהירה יותר של נתונים יתרונות: מהירות גבוהה יותר של העברת נתונים ומהירות חיוג ומענה גבוהים . *ADSL*: גם על גבי קו טלפון אנלוגי, מהירות אסינכרונית בין *upload* ל-*download*. כבלים: אותו רעיון על תשתית אחרת, אך כמו שידורי הוידאו, גם התקשורת משותפת בין משתמשים על קווים אלו ולכן יכולה להסתם

פרוטוקול נפוץ לשכבה פיסית בשימוש במודם : *PPP*: *point to point*. צורת ההתחברות: תוכנה מורה על פעולת חיוג במודם , ולאחר יצירת ערוץ תקשורת מופעלת תוכנית (לא בהכרח מנהל התקן) ליישום הפרוטוקול. מבצעת אימות זהות ומקבלת פרמטרים להעברת מנות *IP* בערוץ, ואז מעבירה מנות *IP*. השליחה: העברת מנת *IP* למנהל התקן *PPP* (חלק ממערכת ההפעלה), מעביר לתוכנית ששולטת במודם ומשם למודם . בצד השני של הספק העסק דומה. פרוטוקול ה-*PPP* גמיש ויכול לעבוד קצת אחרת.

להעברת נתונים למרחקים ארוכים ובמהירות גבוהה צריך*WAN* (*wide area netwok*), רשתות המשמשות לארגונים גדולים.

7.2 שכבת הרשת: ניתוב מנות *IP*:

מטרה עיקרית: ניתוב מנות בין מחשבים שלא מחוברים ישירות. ברשתות *IP* ניתוב מנות מתבצע כך: נסתכל על התקשורת כגרף בו מחשבים הם צמתים וערוצי תקשורת הם קשתות. לכל קצה יש שם ייחודי, זוהי כתובת ה-*IP*. בשליחת מנה מצויינת כתובת ה-*IP* אליה נשלחת (מייצג את המחשב אליו רוצים לשלוח). מסלול המנה נקבע באופן דינמי כך שכל צומת (מחשב) בדרך מחליט את היעד הבא במסלול . כלומר המחשב השולח לא מחליט את כל המסלול אלא רק את הנקודה הבאה בו וערוץ התקשורת בו תעבור המנה אליו . כך עד שהמנה תגיע ליעדה . בגרסת *IPv4* כתובות הן בנות *32b* ונכתבות ע"י 4 מספרים עשרוניים עד *256*. *IPv6* גרסה חדשה, אך לא בשימוש נרחב. *IP* אינו אמין, גם אם בנוי על שכבת מי קשר אמינה: למשל יכול לקבל מנות מקו מהיר (אתרנט למשל) ולשלוח בקו איטי (מודם), ובגלל הבדלי מהירויות ישלח רק חלק מהמנות שכבת ההעברה מתגברת על כך (בהמשך).

מנגנון קבלת ההחלטות ברשת *IP* מבוסס על הקצאת כתובות *IP* בצורה אינפורמטיבית לקבלת החלטות במבנה הנתונים טבלת ניתוב (*routing table*). בטבלה זו מופיעות תחיליות כתובות *IP*, ולכל תחילת מהי הקשת הבאה במסלול של מנות המיועדות לכתובות עם אותה תחילית . בקבלת מנה לשליחה, יחפש המחשב בטבלה את התחילית המתאימה הארוכה ביותר וישלח לפיה . התחילית הריקה היא *default gateway* ונמצאת בכל טבלת ניתוב .

תחיליות מתוארות ע"י שתי מחרוזות *32b*: אחת היא מחרוזת היעד המכילה את התחילית , השניה היא המסכה (*mask*) המכילה 1 במחרוזות התחילת ו-0 בשאר. למשל: תחילית 127 מיוצגת ע"י יעד 127.0.0.0 ומסכה 255.0.0.0 (127 = 8 הביטים הראשונים הם 01111111). הקשת הבאה: שם ממשק תקשורת בשכבת המיקשר, הצומת הבא: המחשב הבא הקרוי *gateway*, מיוצג ע"י כתובת *IP*.

רוב המחשבים, כמו הביתיים, מחוברים רק למחשב אחר אחד. מחשבים אלו בעלי טבלת ניתוב פשוטה: התחילית הריקה, שכן כל מנה ינתבו רק למקום אחד. נתבים לעומת זאת , מחשבים מיוחדים שנועדו למטרה זו בלבד , ובעלי טבלאות ניתוב מורכבות . מנגנון התחיליות מאפשר דחיסת טבלת הניתוב וייצוג החלטות ניתוב למספר רב של יעדים בכניסה אחת בטבלה. לשם כך צריכים להתקיים: 1) החלטת הניתוב לכל היעדים באותה קבוצה היא זהה, 2) לכולם תחילית משותפת *p*, 3) אין יעדים אחרים עם תחילת *p* עם החלטה שונה, אלא אם בעלי תחילת *p+q* לה החלטה נפרדת ספציפית יותר.

כניסה בטבלה עם תחילית קטנה מ- *32b* היא כלל ניתוב, ואם יש כניסה עם הארכה של אותו כלל , היא יוצא מן הכלל. המטרה: טבלה עם מעט כללים ומעט יוצאים מן הכלל, וזה תלוי באופן הקצאת כתובת *IP*. הכתובות מוקצות באופן מאורגן והיררכי. ארגון עולמי מחלק לגופים גדולים מרחב כתובות מסוים. למשל אוני' ת"א היא מרחב הכתובות שמתחילות ב-132. גופים אלו יקצו תתי תחומים לתתי גופים שלהם וכן הלאה, ויתכן צורך להקצות מספר תתי תחומים, תלוי כמה מחשבים צריך לכסות. הקצאת תחומי הכתובות באופן היררכי צריכה להתאים למבנה ההיררכי של הרשת ולא למבנה ההיררכי של חברות וארגונים. למשל חברה שיש לה שני אתרים בני מקומות שונים, צריכה לשמור על אחידות תחילת לשניה האתרים (גם אם אלו מחוברים דרך ספקים שונים), וזאת כדי שיהיה כלל ניתוב יחיד בטבלאות הניתוב המובילות לחברה. טבלאות ניתוב צריכות להגדיר מסלולים ללא מעגלים כדי למנוע אי הגעת מנה ועומס מיותר. מנה מחזיקה את מספר המחשבים שעברה, וכך ניתן להפטר ממנות במסלול מעגלי.

7.3 ממשק שירותי התקשורת: שקעים:

ממשק שירותי תקשורת מבוסס על שקעים (*sockets*), לקישור תהליך לשכבת ההעברה . קריאת המערכת *socket* יוצרת שקע שהוא ערוץ תקשורת וירטואלי בין תהליכים, ועוד לא מחובר. הקריאה היא עם שני פרמטרים: משפחת הפרוטוקולים (*IP* למשל) וסוג הקשר. יש שני סוגים חשובים למשלוח רצוף וסדור של *datagram*. השמות שניתנים לשקעים קרויים *port* (ביוניקס: מסלול במע' הקבצים). מתן שם (*port*) באמצעות *bind*. ברוב המקרים הקמת קשר היא בשיטת שרת- לקוח. בשיטה זו: השרת יוצר שקע ומבצע *listen*, הקמת ערוץ תקשורת . לאחר מכן *accept* גורמת לתהליך להמתין להקמת קשר דרך השקע. הלקוח קורא ל-*connect* ומקים קשר עם שקע השרת, *accept* מחזירה לשרת שקע חדש מחובר לזה שהקים הלקוח. כעת ניתן לתקשר ע"י *read/write*. השרת יכול לקרוא שוב ל-*accept* להקמת קשרים נוספים. *Listen* הוא רק ערוץ הקמת קשר, לא קיום התקשורת עצמה.

הערה: *socket, accept* מחזירות מזהה שתקף רק בתהליך, כמו *open* לקובץ, ואין להם קיום מחוץ לו. קריאת *bind* קושרת למזהה חיצוני, *port*.

סוגי התקשורת: ב-*UDP* מנות נשלחות ל-*IP,port* מסויימים ללא וידוא הגעה ליעד , חיסכון בתקורה אך אובדן אמינות . ב-*TCP* שני פורטים משני *IP* מתחברים לזמן קבוע, *session*, ונעשה מימוש תקשורת אמינה עם וידוא הגעת מנות שנשלחו, על חשבון תקורה גבוהה.

<u>7.4 זרימת מנות במחסנית הפרוטוקולים</u>:

להלן תיאור תהליך שעוברת מנה: מנה מתהליך עוברת למערכת ההפעלה דרך שקע שנקשר ע"י *socket* לפרוטוקול בשכבת ההעברה (*TCP, UDP*). לפי מבנה הנתונים של השקע מזוהה הפרוטוקול ונקראת השגרה המתאימה לו , שמוסיפה למ ידע תחילית בהתא ם: יעד (*IP* ושער). זו שגרה של שכבת ההעברה, ומעבירה את הנ"ל לשכבת הרשת (*IP*). שכבת הרשת מוסיפה תחילית משלה: כתובת מחשב היעד, מזהה פרוטוקול ההעברה (שבצד השני ידעו לאיזה פרוטוקול להעביר). ב-*IP* שכבת הרשת מחליטה איך לנתב המנה . המערכת מחפשת את הכניס ה בטבלת הניתוב עם התחילית הארוכה ביותר המתאימה. לפי הכניסה ידועה תווך התקשורת (שכבת המקשר) ומחשב היעד (כתובת *IP*) הבא במסלול (אם הבא הוא היעד הסופי , לא תצויין כתובתו בטבלה). לאחר החלטה לפי טבלת הניתוב, המנה עוברת למנהל ההתקן שטבלת הניתוב בחרה, למשל של כרטיס אתרנט עם *IP* יעד (הבא במסלול) ותווך. מנהל ההתקן מתרגם כתובת *IP* לכתובת כרטיס אתרנט ייחודית לכל כרטיס כזה בעולם (*MAC address*). מנהל ההתקן שומר לרוב טבלת תרגום של תרגומים אחרונים. אם הכתובת לא בטבלה זו, שולח מנה בפרוטוקול *ARP*: כל המחשבים במקטע האתרנט מקבלים, וזה שה-*IP* מתאים לו שולח חזרה מנה עם הכתובת הפיסית שלו. ניתן לנהל טבלת תרגום זו ע"י *arp*. מנהל ההתקן מוסיף למנה תחילית הכוללת את פרוטוקול שכבת הרשת (*IP*), כתובת פיסית וקוד גילוי שגיאות. אז מורה לבקר לשדר את המנה.

בצד השני: בעת קבלת מנה ע"י מנהל ההתקן של שכבת המיקשר, מגלה לאיזה פרוטוקול שייכת. למשל אתרנט יזהה מנת *ARP* שצריך לענות לה. מנה מפרוטוקול *IP* תועבר לטיפול שגרה בשכבת הרשת לאחר הסרת תחילית שכבת המקשר. ב-*IP* יזוהה פרוטוקול שכבת ההעברה המתאים, תוסר תחילית ה-*IP* ותועבר המנה לפרוטוקול ההעברה המתאים (למשל *UDP*). שם יזוהה ה-*port*, יוסר ויועבר המידע במנה לתהליך שקורא מהשקע שקשור ל-*port*.

<u>7.5 תרגום כתובות והתחפשות</u>:

מחשב שמנתב כתובות *IP* מסוגל לשנותם, למשל לאפשר למחשב אחד להתחפש לאחר . מנגנון זה, *NAT*, מאפשר לארגון גדול שימוש במספר כתובות קטן. מחשב המחבר ארגון לעולם צריך להיות בעל כתובת נגישה וידועה לכל מחשב באינטרנט, אך כתובות שאר מחשבי הארגון חסרות משמעות לעולם. כתובות אלו ניתנות לשימוש פנימי בכל ארגון. המחשב המתרגם (הקשר לעולם) מקבל מנה ממחשב בארגון, משנה כתובת המנה לכתובת שלו ואת השער לשער חדש שלא בשימושו, שומר את התרגום הזה בטבלה, ושולח. כשחוזרת תשובה היא ממוענת אליו, ואז לפי הטבלה מתרגם את ליעד המקורי בתוך הארגון ושולח אליו (ל-*IP* הפנימי והשער המתאים). פעולת התרגום אינה ברמת *IP* בלבד אלא דרוש שינוי גם בתחילית רמת הרשת וההעברה.

<u>פרק 8: פרוטוקול *TCP*</u>:

פרוטוקול ההעברה הנפוץ ביותר מעליו פרוטוקולי יישומים רבים(*HTTP, FTP, SMTP*). תכונות עיקריות של *TCP*:

- תקשורת דו כיוונית אמינה וסדורה מעל פרוטוקול הרשת *IP* שאינו כזה.
- העברה מהירה של נתונים.
- מניעת עומס שעלול להגרם משליחה חוזרת של מנות שאבדו.
- יצירת ופירוק קשר באופן מפורש ומוסכם על שניהצדדים.
- תכונות נוספות: משלוח הודעות דחופות שלא לפי סדר ורצף.

חוסר אמינות *IP* מקשה על מימוש פרוטוקול העברה אמין וסדור:

- <u>קשיים בשמירה על סדר הנתונים</u>: המחשב מפרק את רצף המידע למנות לשליחה, והמחשב המקבל עלול לקבל לא באותו סדר בו נשלחו . למשל: טבלת ניתוב בדרך מתעדכנת למהירה יותר בעת השליחה, מנות אחרונות מגיעות לפני ראשונות.

- <u>קשיים בהבטחת העברה אמינה של מידע</u>: מנות עלולות ללכת לאיבוד: 1) *IP* אינה רשת אמינה. 2) גם אם הגיעו מנות, אולי יאבדו מחוסר מקום לאחסנם. לכל שקע *TCP* יש במחשב המקבל חוצץ זמני אך זה עלול להתמלא. לשכבת ההעברה אין דרך לגרום ליישום לרוקנו ולפנות מקום חוסר סדר וחוסר אמינות דורשים מספור המנות בסדר שליחתן ומנגנון שליחה חוזרת של מנות שלא התקבלו אחרי פרק זמן סביר, מותנה באישור המקבל. ה-*TCP* משתמש בחוצצים לצד השולח ולצד המקבל, ואלו משפרים את יעילות העברת המידע ו התוכניות המחוברות לשקע ה- *TCP*. <u>בצד השולח</u>: ללא חוצץ שליחה היתה רק כשהתהליך היה רוצה לשלוח, וכשהרשת היתה פנויה (לא שולחת מנה אחרת). שני תנאים אלו גורמים חוסר יעילות: 1) לא ניתן לנצל זמנים שהרשת פנויה. 2) התהליך השולח היה נתקע. החוצץ מאפשר לתהליך לרוץ ברצף ולש כבת ההעברה לצבור מידע ולנהל ביעילות את הרשת. <u>בצד המקבל</u>: בלי חוצץ מידע היה מתקבל רק אם התהליך היה במצב קבלה (קריאת מערכת *read*), בזבוז משאבים. היה לפיכך צורך לשלוח מנות שוב ושוב כדי לודא שאכן המקבל ישמור אותן. לכן ב-*TCP* יש בקרת זרימה: הצד השולח שולח רק כשיוד ע בודאות שלמקבל יש מקום פנוי. אפשרות נוספת ללא חוצץ: המקבל יזום הודעה לשולח שיש לו מקום, אך אז היה עיכוב בשליחה עד קבלת אישור. סיכום צורך בחוצצים: 1) ניצול זמנים בהם הרשת פנויה ולא כתלות בתהליך, יעילות תהליכים מחוסר צורך לחכות לרשת, כתיבה/קריאה מיד מהחוצץ.

<u>8.1 חלונות מחליקים (*sliding windows*)</u>:

הרעיון: מידע מיוצג כרצף אינסופי של בתים, המקבל מעביר לשולח מידע ממנו יכול להסיק על קטע שטרם קיבל ויש לו מקום בשבילו. קטע זה הוא חלון, ומחליק - משום שמתקדם עם התקדמות הרצף. תחילית ב-*TCP* מתארת את מצב הפרוטוקול בצד השולח. שדות עיקריים:

- כמות המידע (בבתים) מתוך רצף הנתונים שנשלח במנה. יכול להיות 0.
- מיקום המנה ברצף מודולו 2^{32} ב-*32b*, וזאת ביחס למספר אקראי התחלתי מוסכם על שני הצדדים בתחילת התקשורת (למנוע התחזות צד שלישי).
- מיקום הבית האחרון שנקלט ברצף שמקבל השולח מהמקבל מודו לו 2^{32}. זהו אישור הקבלה שמודיע למקבל אילו נתונים התקבלו בהצלחה, כדי שידע לא לשלוח נתונים אלו יותר. ה-*TCP* לא מאשר מנות, אלא נתונים.
- גודל חלון: מודיע כמה בתים שטרם אושרו מהמשך הרצף, חוצץ הקבלה יכול להכיל. הצד השני ידע לא לשלוח יותר נתונים מגודל החלון.
- סכום בדיקה: מאפשר למקבל ביטחון גבוה שהמנה התקבלה על ידו ללא פגיעה בנתונים
- שדות נוספים פחות חשובים, כגון: ציון מצבים מיוחדים, הסכמת פרמטרים שקשורים להתקשרות (כמו צורת ייצוג גודל חלון).

גודל החלון שמקבל השולח מגדיר לו טווח בחוצץ שממנו יכול לשלוח נתונים, בידיעה שלמקבל יש מקום לקלטם. הנתונים ישארו בחוצץ כל זמן שלא אושרו. כשיאושרו יצאו מהחוצץ והחלון יחליק קדימה, ועוד מידע בחוצץ יהיה רשאי להישלח. בצד המקבל יש נתונים שאושרו (סדר ואמינות) ונקראו מהחוצץ, ויש חלק שאושרו ועדיין בחוצץ (טרם נקראו ע"י התהליך). בסוף הקטע המאושר עד סוף החוצץ – זהו גודל החלון. אם יש חור בחלון, עד שלא יושלם לא ניתן יהיה לאשר את הנתונים שאחריו. דוגמאות לתגובות לאירועים:

- <u>המקבל מקבל מנה שממלאת את החור</u>: סעת קבלת השלמה לחור, מעביר לשולח מנה המאשרת את כל הרצף עד כה, כולל המידע שהיה בחוצץ אחרי החור. בתחילית זו יוכרז על גודל חלון קטן יותר (החוצץ של המקבל התמלא עוד נתונים מאושרים). לא בהכרח תשא מידע נוסף לתחילית. מיד עם השליחה השולח עוד לא קיבל את האישור, וגודל החלון בו נשאר אותו דבר בינתיים.
- <u>השולח מקבל אישור הקבלה</u>: ונעשה סנכרון בין השולח למקבל. השולח מוחק את הנתונים שאושרו מהחוצץ שלו.
- <u>התהליך המקבל קורא נתונים מהחוצץ</u>: נתונים הועברו מהחוצץ לזיכרון התהליך, ואלו נמחקים מהחוצץ. אותה מנת אישור שנשלחה קודם נשלחת שוב עם עדכון גודל החל ון שגדל כיון שכמות הנתונים המאושרים בחוצץ קטנה. אם התהליך המקבל מנסה לקרוא יותר נתונים מאשר יש בחוצץ, יקבל את מה שיש ויאלץ להמתין עד קבלת השאר.
- <u>השולח מקבל את עדכון החלון</u>: השולח יכול לשלוח יותר נתונים משאושר לו קודם והם ישארו בחוצץ עד אישורם מהמקבל.
- <u>התהליך השולח כותב נתונים לחוצץ</u>: אם כמות הנתונים לשליחה גדולה מהמקום הפנוי בחוצץ, התהליך יתקע עד שיתפנה מספיק מקום (השולח יקבל אישור מהמקבל על חלק מהנתונים, ואלו ימחקו מהחוצץ של השולח).

<u>8.2 התמודדות עם אובדן מנות</u>:

כל מנת *TCP* נושאת אישור עדכני וגודל חלון, ועשויה גם לשאת מידע. ה-*TCP* מתמודד עם אובדן מנות בעזרת *timers* (שירותי יקיצה). המערכת מפעילה שגרת *TCP* כל זמן קבוע שבודקת האם פעולה שהיתה חזויה לעתיד צריכה כבר להתבצע אם כן - מבצעת. *TCP* משתמש ב-3 שירותי יקיצה:

<u>Retransmit timers</u>: שידור חוזר, מבטיח שליחת מנה שוב ושוב עד אישורה. כשנשלחת מנה והשירות כבוי, מתזמנים יקיצה לנקודת זמן קרובה. בהתעוררות שולח הפרוטוקול את המנה הישנה ביותר שלא אושרה, ומכפיל את זמן ה המתנה ליקיצה הבאה בקבוע עד חסם כלשהו (כדי למנוע שידור תכוף מדי ועומס) כשהשולח מקבל אישור, מכבה את השירות ומפעילו מחדש (שיתפוס בהתעוררות שוב את המנה הישנה ביותר הלא מאושרת העדכנית). אם אין מנות לא מאושרות, לא יופעל מחדש.

Persist timer: מבטיח שאובדן עדכון חלון בדרכו לשולח לא ימנע מהשולח שליחת מידע נוסף לעד. הצד השולח הוא שיוזם גילוי מנה כזו שאולי אבדה. כאשר לצד השולח יש מידע חדש לשלוח אך החלון סגור /קטן מדי, מתזמן יקיצת *persist* לנקודה בעתיד. בהתעוררות נשלחת כל כמות מאושרת ע"י *H* החלון הקטן, או אם סגור – מנה בת בית אחד. המקבל: אם חלונו פתוח, יצרף את הבית וישלח עדכון חלון חדש. אם סגור, יזרוק את הבית שקיבל אך יבין שזו יקיצת *persist* וישלח לשולח שוב מנה עם עדכון גודל חלון. ביקיצה, השולח יתזמן עוד יקיצה למקרה שגם האישור הבא מהמקבל יאבד. השירות נכבה בקבלת מנה עם גודל חלון שמאפשר שליחת מידע

יתכנו לפיכך קבלת מספר מנות זהות, ועל הצד המקבל להתעלם מתוכנן, אך חייב לשלוח כתגובה מנת אישור וגודל חלון עדכני.

8.3 אופטימיזציות:

עם השנים הוכנסו אופטימיזציות להתמודדות עם חוסר יעילות של הפרוטוקול הבסיסי, למשל התמודדות עם גודל מנה חסום (גודל זה: *MTU*).

1. סינדרום החלון האווילי (*silly-win. Synd.*): כשהחלון קטן מאוד אין טעם לשלוח נתונים כלל. יכול להווצר מחוצץ מקבל מלא, והתהליך המקבל קורא כמה בתים מהקשר, שפותחת חלון קטנטן. שליחת מנות קטנות אינה יעילה ומעמיסה לחינם את הרשת. שיפור: המקבל לא מכריז על חלונות קטנים (מגודל רבע חוצץ עד *MTU*) אלא 0 במקום, השולח מונע שליחת מנות קטנות (פחות מ-*MTU* בלי תחילית) עד יקיצת *persist*. עלול להזיק אם לצד המקבל חוצץ קטן, אז המקבל כן מכריז על חלונות קטנים, והשולח לא ימתין ל-*persist* (הורדת יעילות) אלא ישלח מנה לא מלאה בתנאי שגודלה לפחות חצי מגודל החלון הגדול שהוכרז עד כה (לפני הכרזת הקטן). כך השולח לא ימתין וישלח מיד.

2. השהיית משלוח מנות קטנות (*Nagle's Alg.*): השהית שליחת מידע חדש בחוצץ אם יש בו מנות שעדיין לא אושרו, אלא אם כמות המידע שטרם נשלח ממלאת מנה. אם הכל אושר עד כה, כל כמות שתתקבל תשלח. כשמועבר מידע רב, מנה מצטברת במהירות ונשלחת ללא השהיה משמעותית. כשמועבר מידע מועט: ברשת מהירה אישור מגיע לפני הפעולה הבאה, ולכן גם מידע מועט ישלח. אם אישור מגיע אחרי שליחת הפעולה הבאה, סימן שהרשת איטית, ובמצב כזה מחכים להצטברות מנה – מעמיסים פחות על הרשת האיטית. ניתן לכבות אופטי' זו, שעלולה להזיק במקרים מסוימים.

3. השהיית משלוח אישורים ועדכוני חלון: מנה קטנה זו אינה מכילה חלק מהרצף ועלולה לבזבז משאבים, לכן מושהית עד אחד מהש לושה: 1) חלפה חמישית שניה מאז נוצר הצורך בעדכון, 2) יש מידע לשלוח וניתן לצרף העדכונים למנה זו, 3) חלון הקבלה גדל בינתיים לגודל מנה שלמה. לא משהים יותר מחמישית שניה כדי למנוע בזבוז משאבים מיקיצות מהצד השני של הקשר.

4. קביעת גודל מתאים לחוצצים: חוצץ קטן/גדול מדי בצד המקבל פוגע ביעילות. רוב המימושים משתמשים בגודל קבוע שניתן לשינוי ע"י התהליך. מהו גודל נכון לחוצץ: אם המקבל קורא רציף את הנתונים מהחוצץ, תפקיד חוצץ המקבל להרשות לשולח לשלוח כמות מסוימת של נתונים לפני קבלת אישורם (כדי לא לעכבו). כמות זו שווה לכמות הנתונים שהקשר יכול להעביר בפרק הזמן מרגע שליחת מנה עד רגע קבלת אישור, השווה לכפליים זמן ההשהיה (*latency*), הנקבע ע"י עיכובים במסלול מצד אחד לשני (בגלל טיפול נתבים וכו'). נרצה חוצץ מקבל בגודל מתאים לכמות זו. דוג' בעמ' 148.

חוצץ קטן יביא לחלון קטן ועיכובי שליחה מהשולח, חוצץ גדול יביא לחלון גדול מדי שיאפשר לשולח להעביר כמות גדולה, שתתקע במסלול בדרך בנתבים צרי פס, ויגרום לשליחה חוזרת של מנות שהולכות כך לאיבוד.

5. התחלה איטית (*slow start*): בקשר חדש תתכן בהתחלה הצפת מנות כיוון שהחלון גדול בהתחלה, ולפיכך אובדן מנות בשל עומס בנתבים בדרך. השולח מתחזק לפיכך משתנה חלון עומס המתחיל בגודל מנה אחת, ובכל אישור מנה הוא גדל. השולח לא שולח מידע גדול ממשתנה זה, ומשתנה זה גדל אקספ' החל מתחילת הקשר. כמובן מוגבל עדיין לגודל החלון המוכרז מהמקבל

6. מניעת עומס: רוב המנות אובדות בדרך בגלל עומס בנתבים ופחות מהשחתת נתונים בתווך. ה-*TCP* מסיק מאובדן מנות או השהיות ארוכות על עומס ברשת (לא יכול לדעת אחרת את העומס במסלול או מהו המסלול) ומקטין את קצב השליחה. הבחנה זו באה מפעילות שירותי היקיצה והגעה חוזרת של אישורים. גם מנגנון זה משתמש בחלון עומס לויסות קצב השליחה, וממומש בשולח ללא התערבות המקבל.

7. שליחה חוזרת מהירה (*fast retransmit*): במצב בו אבדה מנה אחת בדרך אך השאר אחריה הגיע למקבל, אין עומס ברשת. הצד השולח יכול להסיק שזהו המקרה באופן הבא: המקבל מפספס מנה, אך בכל קבלת מנה אחרי החור שולח את האישור לבית האחרון לפני החור. כשהשולח רואה 3 אישורים זהים ברצף, מסיק שזה המצב, ושולח את המנה החסרה מבלי להפעיל את מנגנון מניעת העומס.

8.4 הקמת קשר ופירוקו:

קשר *TCP* מוקם ע"י לקוח הקורא ל-*connect* כשבצד השני ממתין *accept*. בשליחת מנה ראשונה אין נתונים ויש ביט מיוחד בשם *syn* שדולק, המודיע על רצון להקים קשר. המנה מכילה מס' אקראי *x* לתחילת מספור הבתים ברצף. מהצד השני תשלח בתגובה מנה גם עם *syn* דולק, אישור קבלה עד מקום *x* ברצף ושליחת מס' אקראי *y* לסימון תחילת הרצף בכיוון השני (משרת ללקוח). הלקוח מתחיל להעביר נתונים עם אישור קבלה עד *y*, והשרת בתגובה מתחיל לשלוח נתונים גם כן. שני האישורים הראשונים – רק סיכנרון הקשר ולא להעברת נתונים.

סגירת קשר: צד שאין לו נתונים לשלוח שולח מנה עם ביט *fin* דולק, המקבל מאשר זאת. קבלת האישור מאפשרת לצד ששלח *fin* לסגור את הקשר, אך הוא עדיין פתוח לשליחת נתונים בצד השני. יסגר סופית כשהצד השני ישלח *fin* וכו'. לאחר סגירה יתכנו מנות שהתעכבו, לכן המערכת זוכרת את הקשר

עוד זמן מה. אם צד מקבל אחרי סגירה *fin*, מבין שהאישור ששלח לא התקבל ושולח שוב. קשר חדש עם פרמטרים זהים (כתובות *IP* ופורטים) יוכל להיות מוקם רק אחרי שהמערכת "תשכח" אותו סופית. פרק זמן זה כמה עשרות שניות, והמערכת מונעת גם הקמת קשר עם אותו מספר שער מקומי. מנות ישנות יכולות לצוץ מהרשת בעתיד למרות המנגנון לעיל. ה-*TCP* מנסה לזהותן: לא מתאימות מבחינת *IP* ופורט לשום קשר פתוח או בעלות מיקום סודר לא מתאים כלל לרצף שאמורות להשתייך אליו. נשלחת כתגובה מנה עם ביט *reset* דלוק, ומקבלה צריך לסגור את הקשר הזה ולהפסיק שימוש בו. כשאין מידע לשלוח יתכן שהקשר לא קיים בלי ידיעת אף אחד מהצדדים (למשל מתקלה או הפסקת קיום מסלול ברשת). יש מקרים בהם התוכנית המשתמשת בקשר צריכה להודיע על כך, כמו מערכת ניטור: צריכה לדעת האם לא מקבלת מנות כי הקשר אבד או כי אין תקלות לקבל. לאפשר הודעה על אובדן קשר משתמשים בשירות יקיצה שלישי: *keepalive timer*. הוא מעיר את הפרוטוקול פרק זמן קבוע אחרי הפעילות האחרונה בקשר, ושולח מנה ריקה. הצד השני יענה באישור או ב-*reset* אם סבור שהקשר לא קיים. אי קבלת אישורים למנה זו גורמת לסגירת הקשר. שירות זה אופציונלי.

פרק 9: מערכות קבצים מבוזרות:

לקבוצת מחשבים שמחוברים לרשת תקשורת מאפיינים שונים מאשר למחשב יחיד:

- רשת התקשורת עלולה להיות איטית משמעותית ביחס למהירות ערוצי התקשורת בתוך המחשב (כמו זיכרון לדיסק).
- במחשב כל הרכיבים פועלים/כבויים (לרוב) יחד, ואילו ברשת חלק יכולים לפעול בעוד אחרים יכבו או יפלו.
- במחשב יחיד הגנת החומרה ע"י המערכת מוחלטת, וברשת מגיעות מנות ממקור חיצוני, ולכן מחשב ברשת לא יתן אמון מלא במנות מהרשת.

לרוב כל מחשב ברשת ירוץ עם מערכת משלו, והמחשבים יספקו שירותים זה לזה. מערכת זו קרויה מערכת מבוזרת. יתרונות גישה לקבצים ברשת:

- קל לשתף נתונים בין משתמשים ותוכניות של מחשבים שונים ומתאפשר בכך פיזור עומס חישובי.
- משתמש יכול לגשת לקבציו ממחשבים רבים שונים אפילו במקומות גיאוגרפים שונים.
- ניהול שרת קבצים מרכזי/צביר שרתים קל מניהול נתונים על מספר רב של תחנות עבודה למשל וידוא גיבויים.
- חסכון כלכלי: 1) שימוש מושכל בשרת מביא לחסכון נפח אחסון, מתורגם ישירות לחסכון כסף. קבצי ונתוני תוכניות רבים משוכפלים על כל מחשב בו מותקנת התוכנית, ושיתוף אלו במקום אחד חוסך מקום. 2) מחיר דיסק אינו לינארי בגודלו, וריבוי תחנות משמעו ריבוי דיסקים גדולים מדי. שימוש בשרת מקצה לכל תחנה את הנפח שהיא זקוקה לו בלבד, וניתן לשתף דיסקים גדולים ויחסית זולים בין כמה תחנות. עם זאת דיסקים לשרתים יכולים להיות יקרים יותר מדיסקים לתחנות עבודה, ולכן יתרון זה אינו קבוע.

9.1 התייחסות לקבצים מרוחקים ופענוח שמם:

שתי שיטות להתייחסות לקבצים מרוחקים: 1) הצבת מע' הקבצים המרוחקת במרחב השמות המקומי, 2) מתן שם לקבצים מרוחקים המשקף את שם השרת ושם הקובץ במרחב השמות של השרת.

9.1.1. הצבה שרירותית של מערכות קבצים מרוחקות:

בגישה הראשונה השרת מייצא את מע' הקבצים שלו/תת מרחב ששורשו מדריך בשרת. כל לקוח יכול להציב (*mount*) מערכת זו אצלו. יוניקס ולינוקס משתמשות בשיטה זו כשפרוטוקול הגישה למע' הקבצים המרוחקות הוא *NFS*, בחלונות בפרוטוקול *SMB* ועוד כמה פחות נפוצים. במערכות חלונות ישנות לא ניתן להציב בכל מקום, אלא רק לאות כונן. יתרון גישה זו: יכולת הצבת המע' המרוחקת בלקוח כך שישקף את השימוש (למשל הצבה ל-*/home/students/cs*). חסרון: לאותו קובץ עלולים להיות שמות שונים במחשבים שונים, עשוי לגרום בלבול ובעיה אם רוצים לאחד מערכות כאלה.

9.1.2. שיקוף שמות שרתים במרחב השמות:

בגישה השניה שם הקובץ מכיל את שם השרת, למשל גישה לקבצים על שרתי *SMB* בחלונות ניתנת ע"י שרשור || בתחילת לשם השרת, ואת זה לשם מערכת הקבצים שהשרת מייצא. למשל, השרת *main* מייצא את *c:\home*. גישה אליו: *\\main\home* היא כאילו גישה ל-*c:* בשרת.

AFS (*Andrew file system*): מע' קבצים מבוזרת שמשתמשת גם בגישה ז ו, בעיקר ביוניקס/לינוקס. גישה לקובץ: תחילית *afs/*, אחריה שם השרת (קבוצת שרתים) ושם הקובץ במרחב השמות (מאפשרת שיתוף מרחב שמות לכמה שרתים). למשל: */afs/tau.ac.il/a.out*.

שימוש ב-*URL*: מזהי משאב אוניברסליים לגישה לאתרי אינטרנט. מכיל את שם הפרוטוקול לגישה לשרת, שם השרת ושם הקובץ במרחב השמות שלו, למשל: *http://www.tau.ac.il/~arielst1*.

יתרון עיקרי לשיקוף: שמות קבועים לקבצים בכל הלקוחות. יתרון משני: הצבה במרחב שמות מקומי דורש הרשאות מנהל מערכת, בעוד שיקוף לא. כך יכול כל משתמש לגשת לקבצים בשרת אליהם יש לו הרשאות בלי התערבות מנהל מערכת. החסרון העיקרי: לקבצים שמות ארוכים שלא משקפים את השימוש בהם, ניתן להתגבר על כך עם מצביעים סימבוליים. שיקוף מיקום פיסי לא תמיד חסרון, כיוון שכך המשתמש מודע למיקום הפיסי של הקבצים, גם במקרה וינותק. חסרון נוסף הוא שהעברת תתי מרחבים משרת לשרת יכולה לשנות שמות קבצים. עם זאת ניתן להחליף שרת ללא שימוש שם הקבצים כי שם שרת מתורגם לכתובת *IP*, והיא קשורה למחשב מסויים, ע"י שינוי מיפוי שם לכתובת או כתובת למחשב ניתן לשמור על שמות הקבצים

תוך שינוי השרת הפיסי. זה לא ישים בפיצול שרת לשני שרתים או יותר. ב-*AFS* בה שם שרת הוא למעשה קבוצת שרתים שניתן לנהל את מרחב השמות שלהם, זה ניתן (כל עוד העברת קבצים היא בתוך אותה קבוצה).

בחלונות ניתן לגשת לקבצים דרך שם שרת או הצבת מע' קבצים מרוחקת, המשתמש בוחר דרך הנוחה לו.

9.1.3. פענוח שמות קבצים מרוחקים:

בפתיחת קובץ מרוחק המערכת צריכה לפענח את שמו למזהה פנימי שיאפשר גישה. הפענוח מתחיל כמו קובץ מקומי, וכשמגיע לתחילית שמייצגת נקודת הצבה של מע' מרוחקת/תחילית כללית של מ ע' מרוחקת, הפענוח עובר למודול תוכנה שמטפל בקבצים מרוחקים בפרוטוקול המתאים. למשל אם הפרוטוקול הוא *NFS*, הפענוח יעבור למודול שמטפל ב-*NFS*. הפענוח יועבר לשרת וזה יחזיר מזהה פנימי ללקוח, שונה לרוב מהמזהה הפנימי של הקובץ בשרת (למשל *inode*). בגישות הבאות לקובץ הלקוח ישלח לשרת מזהה זה. תהליך פענוח בחלונות עבור מע' הממופות לאות כונן דומה.

בשרתי *AFS* ו-*SMB* הפענוח קצת שונה: בזיהוי תחילית שמתייחסת באופן כללי לקבצים מרוחקים בפרוטוקול מסויים, האחריות עוברת למודול המתאים. ב-*AFS*: לאחר פענוח התחילית */afs*. ב-*SMB*: אחרי פענוח התחילית \\. המודול מפריד את סיומת שם הקובץ משם השרת ושם הקובץ בשרת, ושולח לשרת המתאים.

9.2 הטמנת נתונים והסמנטיקה של גישה לקבצים:

אם כל גישה לקובץ מרוחק תתבסס על תקשורת עם השרת, הביצועים יהיו גרועים ממע' קבצים מקומית: 1) תקשורת איטית לעומת גישה מהירה לדיסק מקומי, 2) עומס על השרת כשמשרת כמה לקוחות. גישה דרך רשת תהיה לרוב איטית מגישה מקומית, אך לא בסדר גודל. עומס לקוחות על שרת גורם לחלוקת ביצועיו בין רבים, ובקשת לקוח עלולה לחכות זמן רב למענה בשל עומס על גישה לדיסק (מצד שני לשרת מערך דיסקים, גישה מהירה מאשר לדיסק אחד מקומי). גם בקר התקשורת של השרת משרת לקוחות רבים. פתרון: חיבור השרת למתג מהיר ממתג הלקוחות (*1Gb* לשרת לעומת *100Mb* ללקוחות). פתרון בעיית עומס על השרת ותקשורת איטית: *caching* זמני בלקוח המונע צורך תקשורת לכל פעולה על קובת מרוחק כך פעולה על קובץ שמועתק מקומית (לזיכרון/דיסק) מהירה וחוסכת גישות לשרת, וגם כתיבה לשרת נדחית לשלב מאוחר (כתיבה מקומית זמנית), ונראית מהירה (כי כותבים מקומית) וחוסכת עומס על השרת. בעיה: שינוי סמנטיקה של גישה לקבצים לעומת מערכת קבצים מקומית. הקבצים מהווים זיכרון משותף למחשבים רבים. סמנטיקת גישה לקבצים מקומיים ביוניקס/לינוקס:

- פעולות קריאה/כתיבה הן אטומיות. משמעות: העברת נתונים אל/מקובץ נראות כאילו נעשות בבת אחת ולא הדרגתית.
- קריאה רואה את כל הכתיבות שנעשו לפניה.
- פעולות מסויימות גורמות לשמירת נתונים במדיה יציבה, למשל סגירת קובץ: קריאת המע' חוזרת רק אחרי סיום כתיבת נתונים לדיסק.

במערכת קבצים מבוזרת עם הטמנת נתונים תכונות אלו לא נשמרות. אם נתונים שנכתבים למטמון לא נשלחים אטומית לשרת, קריאה וכתיבה לא יהיו פעולות אטומיות. לעתים יש ויתור על תכונות אלו למטרות ביצועים גבוהים. יש מערכות שממשק שרת לקוח ברמת בלוקים בדיסק, ולשרת אין ידע על אירועים ברמת קריאות מערכת על קובץ אצל הלקוח, רק פעולות על בלוקים בדיסק, ולא מאפשרות לכמה לקוחות לגשת לאותם נתונים. קיצוניות שניה: אי הטמנה מקומית, ביצועים איטיים עקב רשת איטית ועומס על השרת

<u>פתרון מערכת *AFS*</u>: *session semantics*; קובץ שנפתח מועתק בשלמותו ללקוח, ובסגירה מעודכן בשרת רק אם השתנה. כל עוד הקובץ לא השתנה אצל הלקוח, לא יועתק שוב בפתיחה הבא ה (תעשה סה"כ בדיקת עדכניות מול השרת בפתיחה). כל פעולות לקוח מפתיחה עד סגירה הן אטומיות ל אחרים. יתרון: ביצועים טובים אם קוראים את כל הקובץ לאחר פתיחתו. השרת קורא ושולח בבת אחת (ולא מנות קטנות כל פעם), וההטמנה משפרת מהירות מקומית וחוסכת עומס על הרשת. יתרון נוסף: קל לממש את המודול במערכת של הלקוחות שמאפשר גישה לקבי מרוחקים, סה"כ שינוי *open,close*. כל שאר הקריאות ממשיכות לעבוד מול קבצים מקומיים. חסרון: הנחת קריאת קובץ במלואו לרוב נכונה, במסדי נתונים אין זה נכון, והביצועים רעים.

<u>פתרון מערכת *NFS*</u>: מע' הקבצים המבוזרת הנפוצה ביותר ביוניקס/לינוקס. הלקוח שומר מטמון בלוקים בגודל קבוע (לרוב *8KB*) מתוך קבצים. ב-*NFS* לא מובטחת אטומיות כי מעבר בין שרת ללקוח מתבצע בבלוקים בגודל קבוע, ולא מבטיחה עדכניות הנתונים אצל הלקוח. למרות סמנטיקה חלשה, ביצועיה טובים בגלל הטמנה אגרסיבית. כיוון שרוב השימוש המרוחק מלקוחות מבודדים, בעיות הסמנטיקה לא משפיעות. עבודת כמה לקוחות על קבצים משותפים נכשלת בגלל הסמנטיקה החלשה. מערכת *NFS* מתגברת על כך בדרכים שונות.

9.3 הפרוטוקול חסר המצב של *NFS*:

נעשה שימוש בפרוטוקול מיוחד בין לקוחות לשרת כדי למנוע מנפילת מחשבים להשפיע על אחרים. המטרה: נפילת שרת לא תשפיע על לקוחות אם חזר לעבוד תוך זמן קצר, גם אם שכח את כל המידע על הלקוחות שהשתמשו בו לפני הנפילה. <u>פרוטוקול חסר מצב</u> (*stateless*): השרת עונה לכל בקשה באופן נפרד מבלי לשמור מידע על לקוחות וקבצים פתוחים כנ"ל הלקוח על השרת.

בעת בקשת קובץ מהשרת, הוא מחזיר ללקוח את ה-*inode* שלו כמזהה (כך לא שומר טבלת קבצים פתוחים ללקוחות) וזמן יצירת הקובץ (מאפשר לשרת לזהות בקשות גישה לקבצים ישנים). תהליך בלקוח מתרחש מול מטמון, אלא אם זו קריאת נתונים שלא במטמון. כשלקוח מבקש להעביר מהמטמון לשרת, הוא מעביר את מזהה הקובץ, זיהוי משתמש ומיקום הבלוק המועבר בקובץ. השרת מחזיר אישור שמבטיח שהנתונים הועברו למדיה יציבה

בשרת, אז הלקוח יכול למחוק מהמטמון . בסגירת קובץ : כל הבלוקים מועברים ממטמון הלקוח לשרת , אין צורך להודיע על סגירה (השרת לא מחזיק מידע על קבצים פתוחים). הנ"ל תקף גם על בקשות אחרות , כמו שינויים במרחב השמות , שינוי הרשאות וכו'. כשלא מתקבל אישור מהשרת לא ידוע ללקוח האם בקשתו בוצעה (יתכן שכן , יתכן שמתעכבת ברשת , יתכן שבוצעה והשרת נפל לפני אישור). לכן אי קבלת אישור זמן מה מצריכה שליחה מחודשת. השרת, שלא יודע אם עשה פעולה זו או לא , יבצע הפעולה וישלח אישור . בגלל מנגנון זה רוב בקשות השירות יוגדרו בצורה אידמפוטנטית (ביצוע מס' פעמים של אותה פעולה מניב אותה תוצאה). לא כל הפעולות יכולות להתבצע באופן אידמפוטנט, ויש לקחת זאת בחשבון.

עוד הבדל מול סמנטיקת מערכת מקומית : קובץ נמחק רק אחרי שאין אליו הצבעות ואינו פתוח. כאן יכול להיות מצב בו קובץ פתוח בלקוחות אך נמחק בשרת, וניסיון גישת תהליך בו קובץ פתוח יכשל.

העובדה שהשרת חייב לכתוב נתונים למדיה יציבה לפני שליחת אישור עלולה להביא לעיכוב הלקוח הממתין לאישור . לכך הוספו שתי בקשות שירות : כתיבה אסינכרונית (לא חייבת לכתוב מיד לדיסק) ופעולת *commit* - גורמת לשרת לכתוב לדיסק את כל פעולות הלקוח ש במטמון של השרת. אישור ה-*commit* כולל זהות כל הבלוקים שנכתבו . לקוח יבצע *commint* כשסוגר את הקובץ או כשהמטמון מלא . יתכן שהשרת יפול בין קבלת חלק מבקשות הכתיבה ועד ה-*commit*, ובקשות אלו יאבדו מהמטמון. הלקוח יזהה באישורים מהשרת שחלק מהכתיבות אבדו, וישלח בלוקים אלו שוב.

כששרת *NFS* לא מגיב לבקשות, וכאשר מע' הקבצים מוצבת אצל הלקוח: ב-*hard mount* הלקוח ישלח את הבקשות שוב ושוב עד מענה. ב-*soft mount* יהיה מספר קבוע של חזרות על הבקשה ואז תשלח שגיאה לתהליך המבקש. ב-*interrupt mount* כמו ב-*hard*, אך איתות יגרום לחזרת שגיאה לתהליך. בהצבה קשיחה התהליך נתקע עד מענה השרת , אפשר רק להרוג את התהליך ולא ניתן לעצור את הבקשה . הצבה רכה עלולה לגרום לקריאות מערכת להכשל בשיש עומס . לכן צורת ההצבה לנתונים שניגשים אליהם אינטראקטיבית היא הצב ה ניתנת לפסיקה . הצבה קשיחה טובה למערכות קבצים שהמחשב לא יכול להסתדר בלעדיהם, או מערכות לא אינטראקטיביות, אז הגיוני להשהות את התוכנית עד זמינות השרת מהחזרת שגיאה.

9.4 קונסיסטנטיות בעזרת חוזי-שכירות קצרי מועד:

חוזי שכירות קצרי מועד (*short term leases*): מאפשרים גישה קונסיסטנטית לקבצי *NFS* מכמה לקוחות בלי לפגוע בשרידות של *NFS*. מאפשר לשרת להורות ללקוחות לא להטמין נתונים בקבצים שבשימוש כמה לקוחות. כאן השרת לא חסר מצב, ושומר מידע על פתיחת קבצים. לקוח יכול לגשת לקובץ רק אם יש לו חוזה תקף לקובץ ולסוג הגישה. שלושה סוגי חוזים:

- חוזה קריאה עם הטמנה: מרשה קריאה בלבד והטמנה אצל הלקוח. שימוש כאשר כמה לקוחות קוראים בלבד ולא משנים.
- חוזה כתיבה עם הטמנה: מאפשר קריאה וכתיבה והטמנת נתונים אצל הלקוח. שימוש כאשר לקוח אחד משתמש בקובץ.
- חוזה ללא הטמנה: אסור ללקוח להטמין, כל קריאה/כתיבה חייבת להיות מועברת לשרת. שימוש כאשר כמה לקוחות קוראים וכותבים לקובץ.

לרוב סוגי הגישה הם מהשניים הראשונים , ולכן תהיה הטמנה וביצועים טובים . לקוח שרוצה לקרוא מבקש חוזה קריאה , ואם אין חוזי כתיבה מקבל חוזה קריאה עם הטמנה . מספר לקוחות יכולים להחזיק חוזה כזה על אותו קובץ . לקוח שרוצה לכתוב מבקש חוזה כתיבה , אם אין חוזים על הקובץ מקבל כתיבה עם הטמנה, אחרת יחליף לכל הלקוחות לחוזי כתיבה ללא הטמנה, ולאחר אישור ההחלפה מכולם יעניק כזה גם למבקש הכתיבה.

אם שרת רוצה לשנות ללקוח חוזה אך הוא לא עונה , אז : אם הלקוח נפל, כשיעלה לא יזכור את החוזה, ואין בעיה. אם הוא פשוט לא יצר קשר זמן ר ב, אך עדיין מחזיק בחוזה, עלול להיות בעיה עם הקובץ. לכן כל החוזים קצרי מועד ופגים תוך חצי דקה בערך, ולקוח שרוצה להמשיך להשתמש צריך לחדש את החוזה לפני תפוגה. אם השרת לא מסכים להאריך חוזה עם הטמנה, הלקוח יעתיק את ההטמנה לשרת מהר ככל האפשר. אם שרת לא מצליח לשנות חוזה ללקוח, פשוט ממתין לתפוגת החוזה שלו , וממשיך בידיעה שהלקוח כבר ללא החוזה. השרת מחכה זמן מה אחרי הפקיעה , למקרה שיקבל נתונים מהלקוח אחרי פקיעת החוזה שלו, ולהיות בטוח שהחוזה פקע גם לפי שעון הלקוח. אם שרת נפל ולא יודע איזה חוזים בתוקף אצל לקוחות, פשוט ממתין כמה דקה-שתיים שכולם יפוגו, ואז יכול להתחיל לחלק חוזים חדשים ללקוחות. גם במנגנון זה נשמר מידע על לקוחות ושרתים, אך לאובדן המידע אין השפעות ארוכות טווח ולכן שרידות המערכת לא נפגעת

9.5 אבטחת מידע במערכות קבצים מבוזרות:

בבקשות לקוח השרת צריך להחליט אם להע נות או לסרב, ע"ב מי המחשב המבקש ומי המשתמש המבקש . גישה אחת : לודא שהמחשב מריץ מערכת הפעלה אמינה בעיני השרת, והיא אחראית על ודוא זהות המשתמשים, כך פועלת *NFS*. השרת מחזיק רשימת כתובות *IP* של מחשבים מורשים כנ"ל, ושל פורטים מורשים מתחת ל-1024 (ככאלה הם בהכרח של מ ערכת ההפעלה או יישומי אדמין), ורק בקשות משם יענו. הנחת השרת לא מתקיימת אם גנבי מידע שולחים מכתובות *IP* מורשות ומפורטים נמוכים : פשוט מחברים מחשב (נייד) לכבל שמחובר למחשב מו רשה, עובדים דרך אדמין ולוקחים את אותה כתובת *IP*. או : פריצה למחשב מורשה והרצת תהליכים כא ילו ע"י אדמין. שימוש מתוחכם יותר : פרט לזיהוי *IP* ופורט, השרת יודא את זהות מערכת ההפעלה. למשל, וידוא שהמערכת מחזיקה סססמא כלשהי . חיבור מחשב גנב לא יעבור את מבחן הססמא , אך פריצה ללקוח אמיתי חושפת את השרת לגניבת מידע.

גישה מתוחכמת עוד יותר מבוססת על זיהוי המשתמ ש ע"י השרת. המשתמש צריך להזדהות מול השרת , למשל עם ססמא , רק בתחילת העבודה מול השרת. השרת מעביר בתגובה ללקוח אישור גישה מספרי קשה לזיוף, ובכל גישת לקוח לשרת מעביר לו אישור זה. יתכן ואישור זה יהיה תקף לזמן מוגבל (ארוך). השרת לא צריך כך לסמוך על מערכת ההפעלה ש ל הלקוח. יתרונות : 1) גם אם גנב פורץ למחשב הלקוח , הוא עדיין צריך את פרטי הגישה של

המשתמש כדי לגשת לשרת . 2) המשתמש יכול לגשת לשרת מכל מחשב , ולא רק מחשבים מורשים , כמו גישת מערכת *AFS* (ל-*NFS* אין יכולת זו). גם *HTTP*, שמשתמש ב-*cookies* כאישורי גישה, וגם *FTP* משתמשים בשיטה זו. בעיות נוספות: שימוש במודלי הרשאות שונים ע"י השרת והלקוח.

פרק 10: הגנה ואבטחה:

עקרונות אבטחת מערכות מחשב:

- הגדרות משתמשים ואימות זהות כשהם מתחברים למערכת
- הגדרת הרשאות מתאימות למשתמשים כדי שהמערכת תוכל למנוע גישה אסורה למידע ומשאבים
- מודעות משתמשים לפעולות של תוכניות, למנוע פעולות מזיקות שלא ביודעין.
- הצפנת מידע.

10.1 אבטחת מערכות מחשב ופריצה אליהן:

מערכות מחשב חשופות לשתי תקיפות עקריות: פריצה - תקיפה להשגת תועלת ישירה. תקיפת מניעת שירות – נסיון פגיעה במשתמשי המערכת.

10.1.1. פריצה למערכות מחשב:

מצב בו מי שאינו משתמש לגיטימי מצליח להשתמש במערכת מחשב או משתמש לגיטימי מבצע פעולות שאינו מורשה לבצע נזקים אפשריים:

- קריאת מידע על ידי מי שלא מורשה לכך.
- שינוי/הוספת מידע לקובץ.
- מחיקת מידע.
- שימוש במשאבים כמו כוח חישוב/תקשורת ללא תשלום.

רק לעתים רחוקות פריצות נג רמות משגיאה של מערכת ההפעלה. המערכת מגינה באופן מוחלט על החומרה , ואם הרשתה לתהליך לבצע פעולה , לרוב לתהליך היתה הרשאה (גם אם לא בצדק). פריצות קורות אם כן משלוש סיבות עיקריות:

- <u>גישה פיסית לחומרה</u>: אובדן/גניבת מידע יכולים להתרחש אם פורץ גונב חומרה עם מידע מצוטט לקו תקשורת או חודר למסוף עם גישה חופשית
- <u>התחזות</u>: גניבת זהות משתמש לגיטימי למשל ע"י גילוי ססמא.
- <u>סוסים טרויאניים</u>: תוכנית הגורמת למשתמש לגיטימי לבצע פעולות מותרות לו שאינו מודע להן, המזיקות למשתמש או מועילות לפורץ. המשתמש יריץ אותו כיוון שזהו תהליך מועיל למשתמש וברקע גם מזיק לו, או אפילו תהליך שלא מועיל כלל למשתמש ורק יזיק

הצפנת מידע מקטינה את הסיכון לגניבת מידע פיסית וציטוטים לקווי תקשורת . רוב הוירוסים שמגיעים כתוספת לאימייל הם סוסים טרויאניים מסוג שאינו מועיל כלל למשתמש. פתיחת התוספת מריצה את התוכנית המוסתרת. ישנן דוגמאות לתוכנות של חברות מסחריות שאספו באופן נסתר מידע על המשתמש לצרכי שיווק , ופגעו בפרטיות המשתמשים . סוג נוסף של טרויאני משתמש בפרצה של תוכנה תמימה , שכלל לא פותחה ל הוות נזק. דרכים להקטין חשיפה לסוסים טרויאנים:

- המנעות משימוש בתוכניות ממקור לא ידוע, בפרט תוכנות המגיעות באימייל. לחברות מסחריות גם אינטרס להמנע מסוסים טרויאנים.
- במקרי קיצון ארגונים לא יריצו תוכנות שלא יכולות לבדוק את קוד המקור שלהם
- הרצת תוכנות שרת שעלולות להיות חשופות לתקיפה עם הרשאות מינימליות כך גם אם יותקף, הנזק שיעשה ימוזער.
- ביקורות שמטרתן גילוי סוסים טרויאניים. אלו מתבססות על גילוי קבצים שנקראו/שונו ללא סיבה לגיטימית וגילוי מנות לא לגיטימיות שנשלחות

10.1.2. תקיפות מניעת-שירות:

תקיפות מניעת שירות מזיקות למלערכת המחשב ע "י שימוש אינטנסיבי במש אבים, בד"כ תקשורת. התקיפה לרוב דרך בקשות שירות שלא נדרשות להיות ממחשבים מוכרים למחשב המותקף. שרתי *HTTP* למשל מספקים שירות לכל מחשב ברשת , וניתן להציפם בבקשות, עד כדי קריסת השרת (אם העומס חשף כשלים בתוכנה או מיצוי משאבים). קשה להתגונן בפני תקיפות אלה . ניתן לנסות להתגונן ע "י חסימת *IP* שהתקיף בעבר, הגבלת קצב הבקשות בפרק זמן או האטת השירות ל- *IP* עם בקשות רבות ותכופות . עם זאת, גם אם תהיה התגוננות הרשת עדיין מועמסת , והתוקף יכול לנסות לעקוף את אמצעי ההגנה.

10.2 אימות זהות (*authentication*):

שלוש שיטות עיקריות לזיהוי משתמשים:

- <u>ססמאות</u>: בעזרת ססמא משתמש מוכיח את זהותו, כעקרון פרט מידע שרק המשתמש יודע.
- <u>אתגרים וססמאות חד פעמיות</u>: המשתמש צריך לספק ססמא בהתאם לסדרת ססמאות כלשהי , ולמעשה להוכיח את בעלותו על הסדר ה, למשל בעלות על מחשבון מיוחד המנפיק סדרה זו . אתגרים עובדים על אותה שיטה: למשתמש מוצג מספר ארוך והלקוח צריך לת ת תשובה (מספר ארוך) נכונה, כלומר צריך להחזיק רשימת שו"ת או מחשבון מתאים. התשובה יכולה להיות תלויה גם בזמן.

- <u>תכונות ביומטריות</u>: מחשבים עם אמצעי קלט מתאימים יכולים לזהות טביעת אצבע קול, צורת רשתית.

סססמאות הן אמצעי נוח כי אינן מצריכות מחשבון מיוחד או אמצעי קלט מיוחדים. עם זאת, לא אמצעי אבטחה אמין. אנשים שוכחים, רושמים, בוחרים סססמאות קלות לניחוש וכו'. יש כמה דרכים לחשיפת סססמאות. כאשר סססמא מוסעת ברשת תקשורת באופן לא מוצפן ניתן לפעמים לקלוט אותה . פורץ יכול לקרוא סססמאות מקובץ אם מתחזה למשתמש בעל הרשאות קריאה ממנו . ניתן להשתמש בתוכנות המנסות סססמאות רבות ובמהירות (תלוי האם המערכת מאפשרת נסיונות רבים ותכופים). פורץ יכול להשתמש במידע מאימות הזיהוי , אם זמין, לבצע היקשים על הסססמא . שיטה נוספת היא הרצת תוכנה דומה לתוכנת ההתחברות , והרצתה על מחשב ציבורי , ואם משתמש יקליד פרטים הם ישמרו בתוכנית לפורץ . המשתמש יחזור על פעולתו כי יחשוב שטעה בהקשת הסססמא, כשלמעשה היא נגנבה.

רוב המערכות לא שומרות את הסססמאות אלא הפעלת פונקציות חד כיווניות עליהן. כלומר שבהינתן p יחשבו את $f(p)$ ששמוא אצלן, וקשה למצוא p' עם ערך זהה לפונ' f – קשה לזיוף. פרט לפרק זמן קצר בין הקלדת הסססמא ע"י המשתמש לחישוב ערך הפונ' f, הסססמא לא נשמרת במערכת . כמו כן, מערכות צריכות להגיב לבקשות אימות זהות לאט כדי למנוע ניסוי סססמאות מהיר ע"י תוכנית. לרוב תגובה מהירה לבקשה ראשונה שהולכת ומאטה עם גדילת מספר הבקשות. המערכות יסתירו את $f(p)$ בקובץ לא נגיש למשתמש רגיל, כדי למנוע ניסיון פריצה ע"י פורץ שיודע את f. ביוניקס/לינוקס f היא פונ' ספריה מוכרת, ולכן חשוב להסתיר את $f(p)$. למערכות רבות תוכניות מתוחכמות לשינוי סססמאות , השמות אילוצים על בחירת סססמא ומכריחות שינוי כל זמן קצוב. לחלונות מערכת למניעת התחזות תוכניות לתוכנית ההתחברות: הצירוף *alt+ctrl+delete* יעלה את דיאלוג ההתחברות של המערכת, אך מתוך תוכנית מתחזה יעלה דיאלוג *task-manager*. שימוש במחשבונים ואתגרים לרוב מלווה שימוש בסססמא , אך מאריך ומסרבל את תהליך הכניסה. רעיון דומה: כרטיס חכם, שנקרא ע"י קורא מיוחד ומזדהה אוטומוטית מול המחשב . אימות ביומטרי פחות נפוץ . אימות של מחשב מרוחק מתבצע לרוב ע"י אתגרים וססמאות שיכולים להיות ארוכים(שיח בין מחשבים).

<u>10.3 הרשאות</u>:

עצמים יכולים להיות מוגנים ע"י הגדרת הרשאות , למשל קובץ: כתיבה, קריאה, הרצה (אם זו תוכ נית), ובחלונות גם מנעולים , אירועים, תהליכים (למשל איסור שינוי מצב לאירוע). בד"כ יש משתמש כל יכול(*root* ביוניקס, *administrator* בחלונות, שלא נבדקות לגביו הרשאות.

10.3.1. מנגנונים לייצוג הרשאות:

ניתן לחשוב על הרשאות כמטריצת גישה של משתמשים מול עצמים ברי-הגנה, וערכי המטריצה הם סוג ההרשאות: מה מותר למשתמש לבצע על העצם. המערכת שומרת מטריצה זו בצורה קומפקטית, לרוב לפי שורות – *access control list*. ניתן גם לפי עמודות – *capabilities*, כלומר לכל תהליך נשמרת רשימת הרשאות גישה לעצמים. בזמן גישה המערכת בודקת האעם לתהליך יכולת מתאימה, כנגזרת מזהות המשתמש. יוניקס/לינוקס משלבות שתי גישות אלו (הרשאות מרשימת בקרת גישה או יכולות). יכולות מיוצגות ע"י סססמא שהתהליך צריך לתת לשימוש בעצם או שמורות במערכת ומיוצגות בתהליך כמזהה. יתרון: יכולת היא משתנה רגיל (מוצפן) קל לניוד, חסרון: כל גישה לעצם יקרה. כל מזהה משאב (*handle*) הוא מזהה יכולת. בבקשת גישה המערכת בודקת הרשאות, ואם כן מחזירה מזהה ליכולת, התקף רק באותו תהליך. תהליך לא יכול לזייף יכולת, ולא יכול לשנותה כי במקום לא נגיש לו בזכרון.

במערכות מרובות משתמשים רשימות גישה יכולות להיות ארוכות. טכניקות לדחיסה: 1) הגדרת קבוצת משתמשים ועליה הגדרת הרשאות, משתמשים יכולים להיות בכמה קבוצות. 2) רשימות <u>*allow-deny*</u>: רשימות הרשאה של עצם מרכיבות איברים עם משתמש/קבוצה, סוג הרשאה והאם מותר או לא. בבקשת גישה המערכת סורקת ומחליטה בהתאם. למשל אם ל-א', חבר בקבוצה ק', אסור לקרוא מקובץ ולשאר חברי ק' מותר (ולכל השאר אסור), הרשימה תהיה: א' אסור קריאה, ק' מותר קריאה, כולם אסור קריאה. חוסך צורך בניהול מספר גדול של קבוצות ורשימות בקרה ארוכות. 3) שיתוף רשימות בקרת גישה בין עצמים. לכל עצם נשמר מצביע לרשימת בקרת גישה, לא הרשימה עצמה. חוסך מקום.

10.3.2. מנגנון הרשאות הקבצים ביוניקס ולינוקס:

משתמשות ברשימות *allow-deny* פשוטות. כל רשימת בקרת גישה לכל עצם מתייחסת ל: המשתמש הנוכחי, קבוצה שרירותית וקבוצת כל המשתמשים. לכל אחד מהנ"ל יש שלושה ביטים: *rwx* (קריאה כתיבה וביצוע/פענוח דרך מדריך). יתכן כך פענוח דרך מדריך (גישה לקובץ) אך איסור קריאת תוכנו. בדיקת הרשאות: חיפוש המשתמש המבקש ברשימה הנ"ל לפי הסדר. החיפוש ימשיך עד מציאת הרשאה או עד הסוף. יצירת קובץ יוצרת עבורו רשימת בקרת גישה עם: המשתמש שיצר אותו, קבוצה שרירותית - לרוב הקבוצה המוגדרת ראשית (יחידה) למשתמש אליה שייך, וכולם. ניתן לשנות הרשאות וקבוצה שרירותית ע"י היוצר או משתמש כל יכול.

10.3.3. מדיניות הרשאות:

מדיניות לדוגמא: קבצים של משתמשים בודדים יהיו נגישים רק להם, וקבצים של פקוייקט יהיו נגישים רק לחברי הפרוייקט. ניתן למימוש בעזרת *user-private groups*: מוגדרת קבוצה לכל פרוייקט, וסינגלטון לכל משתמש. הקבוצה הראשית של כל משתמש תהיה הסינגלטון שלו, וברירת המחדל של מסיכת הגישה של כל המשתמשים תהיה 700 (גישה מלאה לו, 0 גישה לכולם). לכל פרוייקט מגדירים מדריך שהקבוצה השרירותית שלה היא קבוצת חברי הפרוייקט וע"י *setgid* מגדירים שכל קובץ חדש תחת המדריך יהיה בעל אותן הרשאות.

10.4 מנגנוני רישום:

רישום נסיונות גישה שנאסרו עוזר לגלות נסיונות פריצה, ורישום נסיונות גישה מוצלחים עוזר לאתר משתמשים המריצים סוס טרויאני. חלונות כוללת לכל עצם בר הגנה רשימת רישום גישה *system access-control list*, הדומה לרשימת בקשת הגישה. כל איבר בה מציין שיש לרשום נסיונות גישה מוצלחים/אסורים של משתמש/קבוצה. ניתן לעקוב כך למשל על נסיונות כתיבה של קבוצה לא מורשית לקובץ מסוים. לא נתמך ביוניקס/לינוקס, אך כן שמורות רישום אירועים חשובים שיכולים להשפיע על אבטחת המערכת.

10.5 הגברה והתחזות מותרת:

מקרים בהם תהליך צריך לפעול עם הרשאות שונות משל המשתמש שמריץ אותו:

<u>עידון מנגנון ההרשאות</u>: דוגמאות למדיניות שלא ניתנת למימוש ע"י המנגנון הסטנדרטי של המערכת: 1) לא ניתן להרשות, למשל, כתיבות חלקיות אך לא אחרות (למשל לאפשר למשתמשים לכתוב ליומן הפגישות שלי, אך רק לאזורים מסויימים ביומן). 2) הצבת מערכת קבצים, שמותרת רק למשתמש *root*: לא ניתן להפריד בין מע' קבצים שרוצים לאפשר למשתמש רגיל להציב (כמו מדיסק נשלף) ובין מע' קבצים שלא רוצים לאפשר זאת.

<u>התחזות</u>: לפעמים רוצים שתהליך ירוץ תחת משתמש אחד אך עם הרשאות של משתמש אחר, למשל מנגנון תזמון תהליכים: שרת יריץ תהליך שמתוזמן תחת הרשאות המשתמש שהזמין אותו, ולא תחת הרשאות *root*.

הנ"ל פועלים בעזרת מנגנוני הגברה (*amplification*) והנחתה של הרשאות. פתרון ביוניקס/לינוקס להגברה: הדלקת ביט *setuid* של קובץ המכיל תוכנית גורם לכך שכל מי שבעל הרשאת ריצה לקובץ, יריצו תחת הרשאות בעל הקובץ. כך ניתן גם לפתור את בעיית ה-*mount* לדיסקים נשלפים (תוכנית שבעליה הוא *root* שמאפשרת *mount* לדיסקים נשלפים בלבד, עם הדלקת ביט ה-*setuid*, יאפשר למשתמשים לבצע את הנדרש).

תהליכים שמריצים תוכניות בהם ה-*setuid* דולק או שה-*root* מריץ יכולים להתחזות. קריאת המערכת *setuid* מעבירה את התהליך למצב התחזות למשתמש אחר, וכך ניתן לעבור מהרשאות המריץ להרשאות בעל הקובץ. תהליכים שרצים ע"י *root* בהתחזות לא יכולים לחזור לאחר התחזות ל-*root*.