

סיכום במערכות הפעלה

מערכת הפעלה היא תוכנית שאחראית באופן בלעדי לכל התקני החומרה כלל המעבד והזיכרון. למערכת ההפעלה 2 תפקידים נוספים: לאפשר לתוכניות להשתמש במשאבי החומרה דרך ממשק אחד ונוח ולנהל בצורה יעילה והוגנת את משאבי המחשב.

הגנה על חומרה

התפקיד העיקרי של מערכת ההפעלה הוא להגן על כל החומרה במחשב מתכניות ולאפשר גישה "מסוננת" בלבד לחומרה. אנו משתמשים במערכת הפעלה כיוון שאנו רוצים להריץ בו זמנית תוכניות שאיננו סומכים עליהן לגמרי. החשיבות העיקרית של ההגנה על חומרה על ידי מערכת ההפעלה הינו תיחום תקלות לתוכנית בודדת.

הגנה על המעבד – מבוסס על הבחנה בין 2 מצבי ריצה: משתמש ומיוחד. במצב מיוחס המעבד מבצע כל פקודה. במצב משתמש המעבד מסרב לבצע חלק מהפקודות.

פסיקה – אירוע חיצוני (בדר"כ) שאינו נגרם כתוצאה מפקודת מכונה של התוכנית. המעבד מגיב לסימון בקווי תקשורת מיוחדים מהתקנים חיצוניים ומגיב בקפיצה לשגרת הפסיקה ומעבר למצב מיוחס.

הגנה על זיכרון – הגנה על זיכרון מתבצעת על ידי שיתוף פעולה בין מערכת ההפעלה ובין רכיבי חומרה שמהווים חלק מהמעבד. המעבד מבצע בדיקת הרשאה בכל גישה לזיכרון באמצעות מבנה נתונים שמערכת ההפעלה מכינה (כתובת המבנה שמור באוגר מיוחד).

שליטה בהתקנים חיצוניים – התקנים חיצוניים נשלטים על ידי **בקרים**. תקשורת בין תוכנית ובקר מתבצעת על ידי קריאה או כתיבה לכתובות מיוחדות. הבקרים מחוברים לערוץ התקשורת המרכזי של המחשב הפס. הערוץ מורכב מקווי נתונים, כתובת וערוץ הבקרה. לכל בקר יש כתובות להן הוא מאזין. אם אחת הכתובות מופיעות בערוץ הכתובת (ואם היא קיימת אז סיבית קלט/פלט דולקת) הבקר יודע שהמעבד מתקשר איתו ומגיב בהתאם. כשהבקר רוצה לזיום תקשורת הוא יכול לעשות זאת על ידי הדלקת סיבית הפסיקה בפס.

ממשקים אחידים לחומרה

כדי לא לכלול בכל תכנית את הקוד המאפשר את היכולת להתממשק למאות או אלפי סוגי התקני חומרה, מערכת ההפעלה מספקת ממשק סטנדרטי היודע לגשת לכל סוג של התקן. כך, כתיבה לדיסק למשל, מתבצעת באופן פשוט ובלי צורך לדעת לאיזה סוג דיסק כותבים. בנוסף, ניתן להכניס לשימוש התקני חומרה חדשים ע"י שדרוג הספרייה בלבד. אם השגרות לא היו חלק ממערכת ההפעלה, היא הייתה צריכה להתערב בכל גישה שלהן לחומרה (תקורה – עשוי להשפיע על ביצועים). כמו כן, מערכת ההפעלה "משדרגת" לעיתים את היכולות של החומרה (לדוגמה: מתן מענה אמין יותר לשרות של קווי תקשורת).

ממשקים של מערכת ההפעלה מספקים שירותים לתוכניות תוך שימוש במספר ישויות מרכזיות:

תהליך – אבסטרקציה של מחשב וירטואלי שבו רצה רק תכנית אחת על ידי משתמש אחד. השינויים היחידים ביזרון מבוצעים על ידי התוכנית עצמה. למחשב וירטואלי זה יכול להיות מעבד אחד או יותר.

חוט – הוא אבסטרקציה של מעבד וירטואלי. כל המעבדים הוירטואליים של תהליך עשויים לרוץ בו זמנית (וייתכן שאכן ירוצו בו"ז במציאות).

קובץ – הוא דיסק וירטואלי. מאופיין בדר"כ ע"י שם (אחד או כמה נרדפים) הנוחים למשתמש. ניתן לכתוב בלוג של נתונים בכל אורך ובכל מקום בקובץ. נתון לבקרת גישה.

קשר – מדמה 2 מחשבים וירטואליים המחוברים בערוץ תקשורת פיזי. מאפשר להעביר מידע בין תהליכים (באותו מחשב או במחשבים נפרדים). מחולק ל-2 סוגים עיקריים: **רציף** (מעביר רצף של נתונים) ו**מנדעים** (מעביר הודעות בדידות).

חלון – מערכת ההפעלה מאפשרת לתהליך בעל החלון להציג בו נתונים. כאשר החלון בפוקוס, מערכת ההפעלה מעבירה נתונים מהמקלדת והעכבר לתהליך ששולט בחלון.

תור הדפסה – מייצג מדפסת. פלט שמיועד להדפסה נכתב לתור על מנת למנוע ערבוב של הדפסות מתהליכים המדפיסים בו"ז. מערכת ההפעלה שולחת את ההדפסות זו אחר זו למדפסת.

ניהול יעיל והגון של חומרה

מנגנון ההגנה על החומרה מאפשר למערכת ההפעלה לנהל את משאבי החומרה כרצונה. היא יכולה לאשר, לדחות או לעכב גישה לחומרה בעקבות בקשה. יכולת זו מטילה על מערכת ההפעלה את האחריות לנהל את החומרה באופן יעיל והגון.

בניהול יעיל אנו שאופים לניצול מרבי של החומרה (עבור התוכניות הרצות על המחשב ולא כתקורה של המערכת). ניהול לא נכון עלול לבזבז משאבים ישירות על ידי מערכת ההפעלה (שימוש רב בזיכרון, מעבד וכו') ובעקיפין (למשל: סדר לא נכון של קריאות דיסק עשוי לבזבז זמן במעבר הראש הקורא בין האזורים השונים בו).

בניהול הגון אנו שואפים שלא יוצר מצב בו תהליך מסוים מקבל אחוז גבוה של זמן מעבד/זיכרון/התקני חומרה אחרים על חשבון תהליכים אחרים (אלא אם הוא חשוב לנו יותר מהם). שאיפה זו חשובה כיוון שלמערכת מחשב יש לעיתים יותר ממשתמש אנושי אחד. בנוסף, גם במחשב עם משתמש יחיד, אנו רוצים ליצור תחושה שכל התוכניות פועלות ומגיבות בו זמנית. ניהול לא הגון של המשאבים, יכול במצב קיצוני ליצור "הרעבה" של תהליך.

מערכות הפעלה נפוצות

רוב מערכות ההפעלה כיום מתחלקות ל-2 קבוצות עיקריות: יוניקס/לינוקס וחלונות.

פוסיקס (Posix) – התקן החשוב מבין הסטנדרטים השונים של מערכות יוניקס. רוב מערכות יוניקס (בפרט לינוקס) משתדלות להיות תואמות לחלוטין לתקן זה. עם זאת, קיימים הבדלים בין הגרסאות השונות (בממשק לתוכניתן, בממשק המשתמש ובעקר בניהול התצורה של מערכת המחשב).

חבילות הפצה – מערכת ההפעלה יוניקס (שהקוד שלה חולק חינם) אינה כוללת רכיבים שונים הנדרשים ע"י הלקוח הסופי כדי להשתמש במערכת (ממשק גרפי, דפדפן וכו'). עם זאת, ארגונים וחברות שונים מפיצים "חבילות הפצה" הכוללות תכונות אלו. רוב חבילות הפצה ניתנות להורדה חינם. חלקן, הן חבילות מסחריות.

בנוסף למערכות הפעלה מ-2 קבוצות אלו קיימות מערכות נוספות ממספר קטגוריות:

- מערכות **למחשבים מרכזיים** – בעיקר מתוצרת IBM וקומפק. מערכות אלו מאפשרות לארגונים להריץ מערכות תוכנה גדולות (ובדר"כ ישנות) ומספקות רמה גבוהה של אמינות.
- מערכות **זמן אמת** – מיועדות למערכות משובצות מחשב (למשל מערכות מוטסות, מערכות שליטה למפעלים/מכונות וכו'). מערכות הפעלה רגילות אינן יכולות להבטיח שמטלות מסוימות יתבצעו תוך פרק זמן נתון. מערכות משובצות דורשות לעיתים זמן תגובה מובטח ולכן זקוקות למערכות מיוחדות. אלו שאינן דורשות זמן תגובה מובטח, יכולות להשתמש במערכות שגרתיות (בדר"כ בתצורה מיוחדת).
- מערכות **למחשבים מעוטי יכולת** (למשל מחשבי כף יד) – למחשבים אלו יש (למשל) כמות זיכרון ויכולת עיבוד מוגבלות ביותר ומערכות רגילות לא מסוגלות לרוץ עליהן (או שהן רצות אך לא מנהלות משאבים ביעילות). לכן, פותחו עבור מחשבים אלו מערכות הפעלה מיוחדות.

קלט פלט

העברת נתונים

ממשקי העברת הנתונים בין החומרה (משאבים) והתוכנה (מערכת ההפעלה) צריך להיות יעיל. בנוסף במקרה של אירוע המצריך תגובה (למשל עכבר), יש לטפל באירוע זה במהירות. אי טיפול מהיר באירוע, עשוי לגרום לאובדן מידע (למשל שחוצץ פקודות העכבר בבקר יתמלא ומידע חדש ייכתב על מידע שלא נשלח) ולחוסר יעילות או תגובה איטית שיגרמו למשתמש תחושה שיש עיכוב בטיפול פעולותיו (למשל הסמן על המסך לא יזוז בסנכרון עם תמונת העכבר).

דגימה – דרך פשוטה לטפל במהירות באירועים. קצב הדגימה חייב להיות גבוה מספיק כדי לא ליצור עיכוב אך דגימה גבוהה מדי מבזבזת זמן מעבד. המנגנון העיקרי בעזרתו מסוגלת מערכת ההפעלה לדגום הוא פסיקה שעון מחזורית. פסיקה זו היא המקום הטבעי בו דוגמת מערכת ההפעלה את ההתקנים שהגיע זמן לדגום שוב. נשים לב שיש בעיה לדגום התקנים הדורשים תגובה מהירה. במערכות הפעלה מודרניות לא שולטים בהתקנים באמצעות דגימה.

פסיקה – מאפשר טיפול זריז ויעיל באירועים. התקן הדורש טיפול יפעיל פסיקה. הפסיקה עוצרת את מה שהמעבד עושה באותו רגע, שומר את המצב הנוכחי (כדי לחזור אליו) ומפעיל **שגרת פסיקה** המטפלת בהתקן. בשיטה זו מתקבל טיפול מהיר באירוע ובנוסף אין בזבז זמן מעבד כאשר אף התקן לא דורש טיפול.

טיפול בפסיקות דורש תשומת לב ל-2 נושאים:

הבחנה בין פסיקות שונות – מעבדים רבים מכילים קו פסיקה יחיד. במקרה כזה, כדי לדעת איזה משאב דורש התייחסות, על המעבד לבדוק את כל המשאבים האפשריים (בזבוז של זמן). כדי לפתור את הבעיה משתמשים **בבקר פסיקות** המכיל קו פסיקה מכל משאב. במקרה של פסיקה הבקר יודע ישר מי הפעיל אותה ומודיע למעבד על פסיקה. בשגרת הפסיקה ישלוף המעבד מהבקר את ההפניה להתקן הדורש התייחסות.

מניעת הפעלה רקורסיבית של פסיקות – בזמן פסיקה, המעבד לא מופעלות שגרות פסיקה חדשות (אסור להפעיל את שגרת הפסיקה פעמיים). הפסיקה החדשה תרוץ כשריצת שגרת הפסיקה המקורית תסתיים. לכן, חשוב שטיפול בפסיקה יהיה מהיר (אחרת התייחסות לפסיקה השנייה תתעכב זמן רב מדי). לשם כך, במקרה של טיפול ממושך, הטיפול יתבצע רק בתום הפסיקה ולא במהלכה. לשם כך, שגרת הפסיקה מכינה מבנה **הפעלה דחוייה של שגרה** המכיל את כתובת השגרה וארגומנטים המוכנס לתור של שגרות דחוייה. בסוף ביצוע הפעולות שלא ניתנות לדחייה, מודיעה שגרת הפסיקה למעבד שניתן לטפל בפסיקות חדשות וקוראת לביצוע תור השגרות הדחוייה. אם הייתה בינתיים פסיקה היא תטופל. אחרת יטופל תור השגרות הדחוייה ולבסוף תתבצע חזרה למקום בו נעצרה התוכנית המקורית. במערכות הפעלה מסוימות (חלונות 2000/NT) פסיקות מסווגות על פי עדיפות בהתאם לכניסה של קו הפסיקה בבקר הפסיקות (או במעבד). פסיקות בקו נמוך מקבלים עדיפות גבוהה יותר. במקרה כזה, פסיקה בקו בקרה נמוך, תשהה טיפול בפסיקה מקו בקרה גבוה.

גישה ישירה לזיכרון – למרות יתרונות הפסיקה (טיפול מהיר וחיסכון במעבד כאשר אין צורך), פסיקות אינן יעילות כאשר צריך להעביר נתונים בקצב גבוהה מאת או אל התקן. במקרה כזה גובה התקורה גבוה מדי ובמקרה וקצב הנתונים קרוב לקצב פעילות המעבד, ייתכן ולא תתבצע העברת נתונים כלל (המעבד לא יספיק). במקרה כזה המעבד מספק כתובת של חוצץ שצריך לקבל את הנתונים בזיכרון ואת גודלו והבקר מעביר את הנתונים ישירות מההתקן לזיכרון (או להפך) ללא התערבות המעבד. זו גישה יעילה מבחינת המעבד. החיסרון בגישה הינה בצורך בבקר מתוחכם שמסוגל לבצע את העבודה. בקר כזה יקר יותר בד"כ מבקר רגיל. כדי להזיל עלות זו, מבצעים שיתוף של **בקר גישה ישירה** בין כמה בקרים אחרים.

מנהלי התקן

מנהלי התקן נוצרו כדי לבודד את רוב חלקי מערכת הפעלה והתוכניות מהפרטים הטכניים של כל התקן. לכל התקן (או משפחת התקנים) יש מנהל התקן משלהם.

ממשק סטנדרטי – למנהל ההתקן יש ממשק סטנדרטי דרכו ניתן להתקשר עם מנהל ההתקן ובאמצעותו עם ההתקן. דוגמה לממשק של מנהל התקן בלינוקס (אין חובה לממש את כל השגרות בכל מנהל התקן):
אתחול ההתקן – באחריות השגרה לבדוק אם ההתקן קיים ולא תחל אותו אם כן.
חיבור וניתוק – בעת חיבור יש לבדוק זמינות ההתקן (ייתכן ותפוס על ידי תהליכים אחרים). שגירת ההתנתקות מודיעה למנהל ההתקן שהתהליך לא זקוק לו עוד. מנהל ההתקן ינסה לרשום את הנתונים שעדיין בחוצץ להתקן וישחרר את השימוש בו לתהליכים אחרים.
Flush – מוודא כתיבה של כל הנתונים שקיימים (אם עדיין קיימים) בחוצץ.
קריאה וכתיבה – קוראות וכותבות מההתקן. הכתיבה לא חייבת להיות מיידית להתקן ועשויה להישמר בחוצץ כדי לשפר ביצועים. מותר למנהל ההתקן גם לספק נתונים מתוך חוצץ.
הזנת מצביע (Seek) – אפשרי רק במנהלי התקן להתקני זיכרון עם גישה ישירה לנתוני ההתקן (למשל דיסקים כן אבל סרטים מגנטיים לא). השגרה קובעת מהיכן בהתקן תבוצע פעולת הקריאה או הכתיבה הבאה.
פעולות לא סטנדרטיות (ioctl) – פעולות ייחודיות למנהל התקן ספציפי. תוכניות שמשתמשות בפקודות אלו תפעל בד"כ עם התקנים מסוימים אך לא עם אחרים.
שגרת פסיקה – נקראת כאשר ההתקן יוצר קשר עם מנהל ההתקן באמצעות פסיקה. שגרת האתחול מעדכנת את מערכת הפעלה באיזה פסיקות יטפל ההתקן ובאמצעות אלו שגרות.
פעולות נוספות – כמו דגימה.

מנהלי התקן עם הטמנה (Block Devices) – הינם מנהלי התקן להתקני זיכרון שמשתמשים במאגר החוצצים המרכזי של מערכת הפעלה (בלינוקס ויוניקס). מנהלים אלו מופעלים על ידי **מאגר החוצצים המרכזי (Buffer Cache)** של מערכת הפעלה. מאגר זה דואג לשמור עותק של נתונים שנקראו או נכתבו לאחרונה. אם נתונים אלו נדרשים בשנית, מאגר החוצצים יספק את המידע בלי לקרוא למנהל ההתקן. הקריאה למנהל ההתקן תתבצע אם הנתונים אינם בחוצץ או אם צריך לכתוב נתונים להתקן כדי לפנות זיכרון.

מנהלי התקן ללא הטמנה (Character Devices) – מופעלים ישירות ללא התערבות של מאגר החוצצים. שולטים בד"כ על התקנים שאינם התקני זיכרון ולכן לא זקוקים לחוצצים (מקלדת, עכבר וכו'). נשים לב שהתקנים אלו עשויים להשתמש בחוצצים אך לא במאגר החוצצים המרכזי. שימוש זה מיועד בד"כ כדי למנוע אובדן מידע (במקרה של הפרשי מהירות בהעברת נתונים). ניתן להשתמש במנהלי התקן אלו גם עבור דיסקים במידה והמשתמש יודע שלא ישתמש בנתונים שוב (ואז פעולת הטמנה היא בזבוז משאבים). שימוש זה נקרא **Raw Access** והוא נפוץ במסדי נתונים.

התקנים ומנהלי התקנים בלינוקס ויוניקס - כדי להשתמש בהתקן חומרה, יש לקרוא לשגרה של מנהל ההתקן (ולהעביר אליה פרמטרים במידה ומנהל ההתקן מטפל ביותר מהתקן אחד). בלינוקס ויוניקס מתייחסים למנהלי ההתקן כמו קבצים (למשל, כדי לקרוא מתקליטור, ניתן לפתוח את `/dev/cdrom` ולקרוא משם נתונים). חשוב לזכור שהשמות מתייחסים להתקני חומרה ולא לקבצים רגילים. כדי לקשר בין השם הזה למנהל ההתקן, בלינוקס ויוניקס קיימות 2 טבלאות (התקנים עם ובלי הטמנה). שגרת האתחול של מנהל התקן, תוסיף מצביע אל מבנה הנתונים שלו בטבלה המתאימה. המיקום בטבלה של מנהל התקן מסוים נקבע בד"כ בעת כתיבת מערכת ההפעלה ונקרא **המספר הראשי (Major Number)** של התקן או מנהל התקן. במידה ומנהל ההתקן שולט בכמה התקנים, לכל התקן יהיה גם **מספר משני (Minor Number)**. כדי לקשר בין מערכת הקבצים ובין התקן מסוים יוצרים **קובץ מיוחד (Special File)** שמציין את המספר הראשי והמשני של ההתקן (בלינוקס עושים זאת באמצעות הפקודה `mknod`). קבצים אלו נמצאים בד"כ במדריך `/dev` אך ניתן לתת להם כל שם. בזמן יצירת הקובץ המיוחד, מערכת ההפעלה אינה בודקת דבר ולכן קיום קובץ אינו מעיד שההתקן קיים/זמין. נשים לב שגם המצב ההפוך אפשרי (קיים התקן ומנהל התקן אך אין קובץ שמצביע עליו). במקרה כזה תוכנית רגילה לא תוכל להשתמש בו (מערכת ההפעלה תוכל באמצעות ציון המספרים שלו באופן ישיר). ישנם מערכות יוניקס ולינוקס מסוימות היוצרות את הקבצים המיוחדים באופן אוטומטי עבור מנהלי ההתקן שקיימים במערכת ושהחומרה שעליהם הם שולטים זוהתה. לשיטה זו קוראים **devfs** ומנהל ההתקן הוא זה שמודיע למערכת ההפעלה מה יהיה שם הקובץ תחת `/dev`.

פקודות מיוחדות בהתקנים - ישנם מקרים בהם יש להעביר למנהל ההתקן פקודות שאינן חלק מזרם הנתונים שאמור להישלח בערוץ התקשורת (למשל, הגדרת קצב העברת הנתונים בערוץ תקשורת טורי, העברת מידע על הפסיקות אליהם מנהל ההתקן צריך להתייחס ועוד). ביוניקס ולינוקס מעבירים מידע זה בעיקר באמצעות פקודת המערכת `ioctl`. קריאה זו קוראת לשגרה באותו שם במנהל ההתקן. היות ולכל מנהל התקן סט פקודות שונה תוכניות המשתמשות בפקודה זו עבור מנהל התקן מסוים לא יעבדו בהכרח עם מנהל התקן אחר.

דיסקים קובעים ותזמונם

דיסק מגנטי מורכב ממספר צלחות המותקנות על ציר משותף ומצופות בחומר מגנטי. הנתונים מאורגנים **במסילות (Tracks)** המחולקות ל**קטעים (Sectors)**. כל קטע מכיל כמות קבועה של נתונים וקוד לזיהוי שגיאות. לכל צלחת יש ראש קורא-כותב. הצלחות מסתובבות בקצב קבוע. הראשים הקוראים-כותבים מותקנים על זרוע שניתן להניע פנימה והחוצה. הזזת הראש נקראת `seek` ואורכת מספר אלפיות שנייה (בהתאם לגודל התזוזה). עם זאת בכל הזזה יש פרק זמן קבוע שדרוש כדי לעצור את הראש ולוודא שהוא במקום. כיוון שבמסילות החיצוניות יש יותר נתונים וכן כיוון שהסיבוב כאמור הוא בקצב קבוע, נתונים מהמסילות החיצוניות מועברים מהר יותר מאשר נתונים מהמסילות הפנימיות. העברת נתונים מהדיסק לבקר/זיכרון מתבצעת בקטעים שלמים. קבוצת כל המסילות שנמצאות זו מעל זו בכל המשטחים של הדיסק נקראת **גליל (Cylinder)**.

כדי לקרוא מהדיסק, יש להזיז את הראש למסילה המתאימה ולחכות לתחילת הנתונים. הזזת הראש לוקחת את הזמן הרב ביותר ולכן השאיפה היא לשמור נתונים שצריך לקרוא ולכתוב בבת אחת בגליל אחד או בגלילים סמוכים. בבקרים פשוטים לא ניתן לתת לבקר פקודה לקרוא קטעים סמוכים. לכן המעבד מחכה עד תום קריאת הקטע הראשון כדי לבקש קריאה של הקטע השני. בזמן שבין סיום קריאת הקטע הראשון לקבלת הפקודה לקריאת הקטע השני, הראש כבר עבר את תחילת הקטע השני ולכן צריך להמתין כמעט סיבוב שלם. כדי להתגבר על הבעיה, ממקמים נתונים בדיסקים אלו **בסירוג (Interleave)**, כלומר, בצורה לא רצופה (למשל כל קטע שני).

במקרים רבים מצטברות בקשות רבות מהדיסק. במצב כזה יש להחליט כיצד לנהל את תור הבקשות. ניהול התור יכול להתבצע במערכת ההפעלה או בבקר עצמו (בבקרים מתקדמים). אלגוריתם הסידור צריך להשיג 2 מטרות (שלעיתים סותרות):
הגינות - לבקשות שהתקבלו קודם יש קדימות ע"פ מאוחרות (בפרט יש למלא את הבקשות - מניעת הרעבה).
יעילות - אנו רוצים למזער את הזמן בו הבקר ממתין להזזת הזרוע או הסתובבות הדיסק (זמן "מת").

אפשרויות הסידור:

FIFO - האפשרות הפשוטה וההוגנת לפיה ביצוע הבקשות נעשה ע"פ סדר קבלתן. אולם זו שיטה לא יעילה כלל ולכן נפסלת.

SSTF - האפשרות היעילה ביותר: בוחרים תמיד מהתור את הבקשה שניתן למלא תוך המתנה מזערית. שיטה זו יכולה לגרום להרעבה ולכן נפסלת.

Scan - שיטת מעלית. בשיטה זו הזרוע נעה פנימה מהגליל החיצוני לפנימי ומשם החוצה בצורה מחזורית. כל בקשה שניתן למלא בדרך מבוצעת. בשיטה זו לא ממלאים אחר בקשות שנוספו לתור לאחר ההגעה לגליל מסוים כדי למנוע הרעבה. זו שיטה יעילה, מונעת הרעבה והוגנת למדי.

c-Scan - שיטה דומה. פחות יעילה אך יותר הוגנת. פועלים בצורה זהה ל-Scan אך משרתים בקשות רק בזמן התנועה פנימה.

c-Look-1 Look - שתי וריאציות יותר יעילות. בשיטות אלו הזרוע נעה בכל כיוון רק עד הגליל הקיצוני ביותר שיש בקשות שמחכות לו (ולא עד הגליל הקיצוני בדיסק).

דיסקים לוגיים ומערכי דיסקים

השיטות הבאות נוצרו כדי לאפשר חלוקה של דיסק לדיסקים קטנים (כדי להפריד בין קבצים למשל) או לחבר דיסקים לדיסק אחד גדול (כדי להשיג קצבי העברה גבוהים יותר או אמינות גדולה יותר).

מחיצות – נתמך ברוב מערכות ההפעלה. המערכת זוכרת איזה גלילים שייכים לכל מחיצה וכך מתייחסת לכל מחיצה כדיסק נפרד. בלינוקס יש לכל מחיצה מספר משני נפרד (הבקר יודע שמדובר באותו הדיסק). טבלת המחיצות (כוללת גודל, מיקום, צורת ארגון הנתונים) נמצאת במקום מיוחד בתחילת הדיסק. ניתן לקרוא אותה מתוך `/proc/partitions`. באמצעות `fdisk` ניתן לבחון את הטבלה ולשנות אותה (גם בלינוקס וגם בחלונות - גם וגם).

דיסקים לוגיים LV – נתמך במערכות לינוקס ויוניקס מסוימות. בשיטה זו כל הדיסקים מחולקים לקטעים גדול קבוע וניתן להרכיב מכל קבוצה כזו דיסק לוגי. 2 יתרונות מרכזיים לשיטה (ע"פ מחיצות): תחילה, ניתן להגדיל ולהקטין דיסק לוגי בקלות. שנית, ניתן להרכיב דיסק לוגי מקטעים ששוכנים על מספר דיסקים.

מערכי דיסקים RAID – משתמשים במספר דיסקים כבהתקן אחסון אחד (ניתן להשיג כאמור גם בדיסק לוגי). בדר"כ מערכי דיסקים מורכבים מדיסקים שלמים ולא מחלקי דיסקים ומסווגים לרמות. כל רמה מספקת שילוב שונה של שירותים.

RAID 0 – מפזר את הנתונים על גבי מספר דיסקים באופן בלוק-מחזורי (הראשון על דיסק 0, השני על דיסק 1 וכך הלאה. לבסוף מתחילים שוב מדיסק 0). גודל הבלוק נקבע בזמן הקונפיגורציה. שיטה זו נקראת גם **שמירה ברצועות (Striping)** כאשר רצועה היא הבלוקים באותו המיקום בכל הדיסקים (למשל הבלוקים במקום 0 שבכל הדיסקים). ברמה זו ניתן לקרוא ולכתוב בקצב גבוה כל זמן שמ"ה נישאת לבלוק שונה מכל דיסק.

RAID 1 – מטרת רמה זו היא להבטיח אמינות. ברמה זו כל בלוק של נתונים נשמר על יותר מדיסק אחד. שיטה זו נקראת **שיקוף (mirroring)**. כך גם אם דיסק אחד מתקלקל ניתן להמשיך לקרוא את הנתונים (כל עוד דיסק אחד לפחות תקין).

RAID 3 – דומה ל-RAID 0 אבל עם שני הבדלים משמעותיים. הראשון הוא שאחד הדיסקים משמש לשמירת הזוגיות של הנתונים בשאר הדיסקים. הזוגיות מספקת **יתירות** כדי לאפשר קריאה גם אם תא אחד נפגע. ההבדל השני הוא בכך שברמה 3 כל הדיסקים חייבים לגשת לאותו בלוק בבת אחת כדי שניתן יהיה לכתוב רצועה שלמה בגישה אחת לכל הדיסקים (אחרת היינו מבצעים יותר כתיבות לדיסק הזוגיות מאשר לאחרים ולכן זמן הכתיבה היה ארוך יותר).

RAID 5 – מספק גם הוא גישה במקביל למספר דיסקים ויתירות אך הוא מאפשר גם גישה לבלוקים בודדים ולא רק לרצועות שלמות. ברמה זו, בלוק הזוגיות של כל רצועה שמור על דיסק אחר (הדיסק הראשון שומר את בלוק הזוגיות של הרצועה הראשונה, השני שומר עבור הרצועה השנייה וכך הלאה במחזוריות). היתרון בשיטה זו הוא שניתן לכתוב במקביל 2 בלוקים מ-2 רצועות שונות אם הם שמורים על דיסקים שונים ובלוקי הזוגיות של 2 הרצועות שמורים בעוד 2 דיסקים אחרים. החיסרון על פני RAID 3 הוא שבכל כתיבה צריך תחילה לקרוא ביט נתונים וביט זוגיות ורק אז לכתוב.

סיבית הזוגיות מאפשרת בדר"כ לתקן שגיאות ולא לזהות שגיאות. עם זאת, נדיר שקורה מצב בו הדיסק פגום אך מציג מידע כאילו הוא תקין (בגלל מנגנון תיקון שגיאות פנימי). במערכות אחרות, ניתן לזהות באמצעות סיבית הזוגיות שגיאות אך במקרה כזה לא ניתן לתקן את השגיאה באמצעותן.

מערכות הפעלה מסוימות מסוגלות לממש את מערכי הדיסקים באמצעות דיסקים ובקרים רגילים (מערכת ההפעלה דואגת לבצע את פעולות טיפול המערך). משתמשים בתצורה זו בעיקר בשרתים קטנים. במערכות גדולות משתמשים בבקרים מיוחדים לצד דיסקים רגילים. בתצורה זו המערך כולו נראה למערכת ההפעלה כדיסק אחד גדול והבקר דואג לטיפול במערך.

התקני קלט/פלט נוספים

דיסקים אופטיים – בתקליטורים המידע מקודד על ידי חורים במשטח. לייזר מקרין אור על המשטח כדי לזהות אם יש במקום חור או לא. מהירות הקריאה הבסיסית של כונני תקליטורים נקבעה על פי קצב ההעברה שדרוש לנגן מוזיקה אך כיום ניתן לקרוא בקצב של עד פי 50. קבצים שמורים בתקליטורים בצורת ארגון שאינה תומכת בשמות ארוכים לקבצים. מערכות ההפעלה מוסיפות את השמות בצורות שונות אך ניתן להכין גם דיסקים תואמים לכל המערכות. יתרונות תקליטורים: הפצה זולה, צריבת מידע על מדיה זולה ועמידות לאורך שנים.

סרטים מגנטיים – המידע שמור על חומר מגנטי המצפה סרט פלסטי ארוך הארוז בדר"כ בקלטת. גישה ישירה למידע איטית במדיה זו (צריך לגלגל את הסרט עד למידע). לכן, השימוש העיקרי הוא לגיבוי של נתונים.

ניהול זיכרון

מנגנון ניהול הזיכרון במערכת ההפעלה נועד כדי ליצור 3 יכולות חשובות:

- היכולת להריץ מספר תוכניות בו זמנית תוך הגנה על הזיכרון של כל תוכניות מפני תוכניות אחרות. יכולת זו היא חלק ממנגנון ההגנה של מ"ה על החומרה (לא ניתן לכתוב לזיכרון של תוכנית אחרת).

- היכולת להשתמש בדיסקים כהרחבת זיכרון. המעבד לא יכול להתייחס ישירות לנתונים בדיסקים כמו לזיכרון. מערכת ההפעלה מחפה על כך ברמת החומרה.
- היכולת להזיז את מבני הנתונים של תוכנית במהלך הריצה שלה מבלי שהתוכנית תהיה מודעת להזזות.

הרעיון העיקרי שמאפשר למערכת ההפעלה להשיג את שלושת המטרות הללו מבוסס על הפרדה בין כתובות שמופיעות בתוכניות ובין הכתובות הפיזיות של תאי זיכרון המופיעות על הפס. כל כתובת שמופיעה בתוכנית מתורגמת לכתובת של תא זיכרון פיזי בכל פעם שהמעבד מתייחס לכתובת. מנגנון זה נקרא **זיכרון וירטואלי** ומאפשר למערכת ההפעלה להשיג את שלושת היכולות הללו.

זיכרון וירטואלי

לכל תהליך יש מרחב זיכרון וירטואלי שגודלו כגודל הזיכרון שניתן להתייחס אליו בפקודות מכונה. הזיכרון הווירטואלי של כל תהליך מחולק לבלוקים בגודל אחיד שנקראים **דפים (Pages)**. מערכת ההפעלה מקצה דפים לתהליך לפני שהיא מריצה בו תוכנית בהם ישמרו קוד המכונה של התוכנית, הקבועים שלה והמחשנית שלה. תוכניות יכולה לבקש הקצאת דפים נוספים אך בכל מקרה יוכל להשתמש רק בדפים שהקצאתם אושרה. ניסיון לגשת לדפים שלא הוקצו לתוכנית ייתפס כ"כ ע"י מערכת ההפעלה שתעיף את התוכנית. הזיכרון הפיזי של המחשב מחולק גם הוא לבלוקים (באותו גודל של הדפים). בלוק בזיכרון הפיזי נקרא **מסגרת (Frame)**. מסגרות אלו יאחסנו דפים של תהליכים. מיפוי דפים למסגרות מתבצע על ידי מבנה נתונים מיוחד. כדי לאפשר לתהליך להשתמש ביותר דפים מאשר כמות המסגרות הקיימות בזיכרון, מערכת הפעלה משתמש בקובץ מיוחד בדיסק הנקרא **אזור הדפדוף (Page/Swap/Backing File)**. אזור זה יכול להיות קובץ, מחיצה או דיסק וגם הוא מחולק למסגרות בגודל דף.

תרגום כתובות

תוכנית שרצה מתייחסת לכתובות במרחב הווירטואלי. כתובות אלו מתורגמות על ידי מערכת ההפעלה והחומרה לכתובות פיזיות שניתן להציג על הפס. התרגום מתבצע פעמים רבות ולכן חייב להיות מהיר ויעיל. בדרך כלל התרגום יבוצע על ידי מנגנון חומרה בשם **TLB**. מנגנון זה הוא חלק מהמעבד והוא מכיל טבל תרגומים של דפים למסגרות בזיכרון. על מנת לבצע את התרגום, החומרה מפרקת את הכתובת הווירטואלית למספר דף (סיביות בכירות) והיסט (סיביות זוטרות). לאחר הפירוק החומרה מחפשת את תרגום מספר הדף הווירטואלי למספר דף פיזי. המספר שנמצא משורשר ביחד עם ההיסט ומרכיב את הכתובת הפיזית הדרושה. כיוון שה-TLB נמצא במעבד כמות התרגומים השמורים ב-TLB קטנה מאוד ביחס לכמות הכתובות המוקצות ע"י מערכת ההפעלה.

חריג TLB – מצב בו התוכנית מבקשת לתרגם כתובת וירטואלית שאינה שמורה ב-TLB. במצב כזה המעבד מחפש את המיפוי בטבלה ששמורה בזיכרון הראשי (מצביע אליה שמור באוגר מיוחד). טבלה זאת נקראת **טבלת הדפים** ואין הגבלה על גודלה (שמורה בזיכרון ולא במעבד). לכן, היא מכילה את המיפוי של כל הדפים לזיכרון הפיזי (מיפוי של דפים לדיסק לא תמיד מופיע בטבלה זו). עם זאת, טבלת הדפים איטית יותר מה-TLB. לאחר מציאת התרגום בטבלת הדפים, המעבד שומר אותו ב-TLB וממשיך כרגיל כאילו הוא בוצע ב-TLB.

מבנה טבלאות דפים

מבנה טבלת הדפים צריך להיות פשוט, יעיל ולצורך כמות קטנה של מקום. מספר מבנים עבור טבלת הדפים:

טבלה שטוחה – טבלה זו היא פשוט מערך שמכיל את כל רשומות המיפוי של המרחב (ע"פ מיקומם במערך). קל לממש אותה אך היא צורכת זיכרון רב ללא קשר לכמות הדפים שהוקצתה.

טבלה היררכית דלילה – שומרת את המיפוי בעץ חיפוש רדוד. לאחר **החלטה על מספר הרמות**, מחלקים את הכתובות ל-2 חלקים. כל חלק מסמן מיקום ברמה. כל רמה (פרט לאחרונה) מכילה מערך של מצביעים בעלי צומת אב משותף. לצד כל מיפוי קיימת סיבית valid המציינת האם לתא זה קיימים בנים או לא. בשיטה זו ניתן לחסוך במקום של מיפויי כאשר תחום דפים שלם אינו מוקצה על ידי טבלת הדפים. היתרון של טבלת הדפים הוא בכך שהיא צורכת פחות זיכרון עבור מרחבים שרק חלק קטן מהם מוקצה (אם כל המרחב מוקצה היא צורכת קצת יותר זיכרון). החיסרון הוא שהיא דורשת מספר גישות לזיכרון כדי למצוא מיפוי (כמספר הרמות בהיררכיה). בנוסף, מנגנון הגישה בטבלה היררכית מורכב יותר. עם זאת, רוב המעבדים כיום משתמשים בה.

טבלה הפוכה – טבלה זו שומרת בטבלת גיבוב (hash) את המיפוי של כל הדפים הממופים ברגע נתון למסגרת פיזית. בנוסף למספר המסגרת הפיזית, שומרת הטבלה את מספר התהליך ואת מספר הדף במרחב הווירטואלי. כדי לחפש מיפוי דף מחשבים פונקצית גיבוב של מספר הדף ומקבלים מספר רשומה בטבלה. אם מספרי הדף הווירטואלי והתהליך תואמים משתמשים במיפוי שברשומה. אחרת, ממשיכים בחיפוש או עוצרים (תלוי מימוש). לטבלה הפוכה 2 יתרונות עיקריים: גודל הטבלה תלוי בגודל הזיכרון הפיזי ולא במספר התהליכים שרצים בו. בנוסף, ניתן להשתמש בטבלת גיבוב שמצריכה מספר קטן של גישות לזיכרון במוצע. החיסרון העיקרי הוא מורכבות מנגנון החיפוש.

תחזוקת טבלאות הדפים וה-TLB

אחריות התחזוקה של טבלאות הדפים שייכת למערכת ההפעלה. היא זו שקובעת לאן ימופה כל דף וירטואלי ובמידה וברצונה לשנות את המיפוי למסגרת חדשה, מעדכנת מ"ה את טבלת הדפים. תחזוקת ה-TLB מבוצעת בשיתוף עם המעבד. מערכת ההפעלה אחראית למחוק מיפויים שאינם תקפים יותר מה-TLB (באמצעות פקודות מכונה מיוחדות). בנוסף, מ"ה צריכה להודיע למעבד מה הוא מרחב הזיכרון הווירטואלי שמשמש את התהליך שרץ כרגע (כתובת הטבלה במקרה של טבלה שטוחה/היררכית ומספר התהליך במקרה של טבלה הפוכה). מעבדים רבים מכילים ב-TLB מרחב של תהליך אחד ולכן בעת החלפת התהליך נמחק המידע מה-TLB. במעבדים אחרים, שמור ב-TLB גם מזהה התהליך ולכן השימוש במיפוי יתבצע רק אם הוא שייך לתהליך שרץ כעת. כפי שצוין קודם, המעבד הוא זה שמטפל בחריג TLB. מנגד, ישנה אפשרות להגדיר שהטיפול בחריג TLB יבוצע באמצעות פסיקה ע"י מ"ה. היתרון בגישה הוא בכך שאין צורך לממש את מנגנון החיפוש בחומרה (מפשט את המעבד). למרות זאת, ברוב המקרים הטיפול נשאר אצל המעבד כיוון שהוא זול יותר. כל הפקודות השייכות ל-TLB וכן הגישה לאזור הזיכרון בו שמורות טבלאות המיפוי זמינים רק במצב מיוחס.

הגנה על זיכרון וירטואלי

ההגנה על הזיכרון מתבצעת באמצעות שיטה זו של תרגום כתובות. למנגנון ההגנה שני חלקים עיקריים:

ראשית, כל התרגומים של כתובות וירטואליות לפיזיות שהמעבד מבצע בזמן שהוא מריץ הליך מסוים מתבצעים ביחס למרחב וירטואלי מסוים. לא ניתן להתייחס למסגרות פיזיות שאינן ממופות למרחב הזיכרון של התהליך. לכן, תוכנית אינה יכולה לגשת למבני נתונים של תוכניות אחרות (אלו שמורים במרחב הזיכרון של התוכנית).

שנית, לכל מיפוי קיימות סיביות המציינות את הגישה המותרת למידע. במידה וקיימת סיבית 1, היא מציינת האם מותר לגשת למידע במצב משתמש או רק במצב מיוחס. לרוב קיימות 3 סיביות המציינות הרשאות כתיבה, קריאה והפעלה במצב משתמש. סיביות אלו מועתקות ל-TLB בעת העתקת מיפוי אליו. יכולת זו של איסור הגישה לדף ממופה מאפשרת למערכת ההפעלה למפות את מבני הנתונים שלה למרחב הזיכרון הווירטואלי של תהליכים רגילים ללא חשש מגישה לא מורשת. ברוב מערכות ההפעלה, מבוצע מיפוי של מבני הנתונים של המערכת לכל מרחבי הזיכרון באותם כתובות. מיפוי זה מאפשר למערכת ההפעלה לפעול בעת פסיקה מבלי לשנות את מרחב הזיכרון. בנוסף, מתאפשר לה להעתיק מידע בזמן קריאת מערכת בין מבני נתונים של התוכנית שרצה ומבני הנתונים של מערכת ההפעלה.

ניסיון גישה שמזוהה כלא חוקי (מיפוי לא קיים, אין הרשאות וכו') ע"י המעבד גורם להפעלת שגרת פסיקה של מערכת ההפעלה. ניסיון גישה כזה נקרא, כאמור, **חריג דף**.

חריגי דף והטיפול בהם

חריג דף הוא מצב בו יש ניסיון גישה לדף שאינו ממופה למסגרת פיזית (ייתכן וממופה לדיסק) או שהרשאות הגישה אליו אינן מספקות. במקרה כזה מופעלת שגרת פסיקה המטפלת בחריג בהתאם לסיבה שגרמה לו.

דף שאינו ממופה במצב משתמש – במקרה כזה תחילה בודקת מ"ה האם הדף הוקצה במרחב הווירטואלי. אם לא, מניחה מ"ה שזו שגיאה ומעיפה את התוכנית (או במידה והתוכנית ביקשה, מודיעה לתוכנית ולא מעיפה אותה). במערכות הפעלה מסוימות, ייתכן שדף הוקצה לתהליך אך לא הוקצתה מסגרת (עד השימוש הראשון). במקרה כזה, המערכת מבינה שמדובר בניסיון גישה ראשון, מקצה מסגרת ומעדכנת את טבלת הדפים של התהליך. בנוסף, ייתכן כי הוקצתה מסגרת אך היא נמצאת בדיסק. במצב זה מערכת ההפעלה משהה את התהליך (תהליכים אחרים יכולים לרוץ בינתיים) וקוראת את המידע מן הדיסק אל מסגרת פנויה בזיכרון הפיזי. כאשר המידע יגיע לזיכרון, מ"ה תעדכן את טבלת הדפים של התהליך שגרם לחריג ותאפשר לו לרוץ שוב. אם אין מסגרת פנויה, על מערכת ההפעלה לפנות מקום בזיכרון (מדיניות הפינוי בהמשך). במקרה של הבאת מסגרות מהדיסק, ייתכן ומ"ה תעדיף להביא גם מסגרות של דפים עוקבים מלבד המסגרת הדרושה כדי לשפר ביצועים. כדי לממש את הפוטנציאל בכך, על מ"ה להקפיד לשמור דפים עוקבים במסגרות רצופות. מאידך, גישה זו עשויה לפגוע בביצועים אם דפים רצופים לא שמורים במסגרות רצופות או במקרה בו אין שימוש לדפים העוקבים לדף שביקשנו (במקרה כזה ייתכן ויוצר חריג דף חדש כאשר נרצה לגשת לדף שבמקומו הצבנו את המסגרות העודפות). כדי להתגבר על כך, מ"ה יכולה להפעיל מדיניות לפיה תתבצע הבאה של דפים עוקבים רק אם יש אינדיקציה לכך שיהיה בהם שימוש בעתיד (למשל, הבחנה כי חריגי הדף האחרונים נגרמו כתוצאה מבקשה של דפים רצופים).

דף ממופה עם הרשאות לא מספקות במצב משתמש – במקרה כזה מסיקה מערכת ההפעלה כי מדובר בשגיאה ומעיפה את התוכנית או מודיעה לה על המצב. מקרים אלו מגנים על ניסיונות גישה שגויים לדפים מוגנים של מערכת ההפעלה וכן בניסיונות גישה לדפים של התוכנית המוגנים חלקית (ניסיון להריץ מידע כפקודת מכונה למשל). הגנה מפני גישה מהווה שרות נוח לתוכניות ומשתמשים לצורך איתור שגיאות תכנות. תוכניות יכולים לאפשר למערכת ההפעלה לזהות שגיאות נוספות על ידי עידון ההרשאות (למשל הסרת הרשאות כתיבה מדפים מסוימים וכו'). מערכת ההפעלה עוזרת לתוכניות לזהות שגיאות על ידי הימנעות מהקצאת הדף הראשון במרחב הזיכרון הווירטואלי, מה שמבטיח חריג דף בכל ניסיון גישה דרך מצביע NULL.

דף שאינו ממופה במצב מיוחס – בדומה למצב משתמש, החריג יכול לקראת כתוצאה מדף הממופה לדיסק, דף שזו הגישה הראשונה אליו או כתוצאה משגיאת תכנות. בשני המקרים הראשונים, הטיפול הזה לטיפול במצב משתמש. במקרה השלישי, על מערכת ההפעלה להבין מה מקור השגיאה. אם מקור השגיאה הוא בקוד התוכנית (למשל העברת ארגומנט לא נכון בקריאת מערכת), הטיפול בשגיאה יהיה זהה לטיפול במצב משתמש. אם מקור השגיאה הוא במערכת ההפעלה, ישנם מספר תרחישים אפשריים: בין קריסה טוטאלית של המערכת להכלה של התקלה והמשך ריצה. בכל מקרה, על מ"ה לזהות את מקור השגיאה ולפעול בהתאם לצמצום הנזק.

דף ממופה עם הרשאות לא מספקות במצב מיוחד – לרוב נוצר כתוצאה משגיאה במערכת ההפעלה אך יכול להיגרם כתוצאה מתקלה בתוכנית שגרמה להעברה של מצביע שגוי.

התייחסות לכתובות פיזיות

לעיתים על מערכת ההפעלה להתייחס לכתובות פיזיות ולא לכתובות וירטואליות. למשל, העברת כתובת טבלת הדפים למעבד או העברת כתובת לבקר גישה ישירה (צריכה להיות כתובת פיזית כיוון שבדרי"כ אין לבקר מנגנון תרגום כתובות). בנוסף, אם הבקשה היא להעביר נתונים ליותר מדף אחד, יש לוודא שהמסגרות ביזרון הפיזי רצופות. ישנם מבני נתונים המאפשרים למ"ה לתרגם כתובות או להקצות רצף מסגרות פיזיות. עם זאת, מספר מערכות הפעלה משתמשות במנגנון פשוט למילוי פעולות אלו: הן מקצות בתחילת המרחב הוירטואלי מספר רב של מסגרות רצופות. **בצורה הפשוטה ביותר, כל המרחב הפיזי מוקצה בתחילת המרחב הוירטואלי.** שיטה זו יוצרת 2 בעיות: מרחב וירטואלי רב "מתבזבז" וטבלת הדפים גדלה משמעותית. על הבעיה הראשונה מתגברים באמצעות הקצאת רק חלק מהמרחב הפיזי לתחילת המרחב הוירטואלי. על הבעיה השנייה מתגברים (במעבדים תומכים) באמצעות התייחסות לדפים מ-2 סדרי גודל. כך ההקצאה בתחילת המרחב הוירטואלי היא של דפים "גדולים" ושאר ההקצאות הם של דפים בגודל "סטנדרטי".

מנגנונים ומדיניות לפינוי מסגרות

הזיכרון הפיזי משמש לא רק כדי לאחסן דפים של מרחבי זיכרון וירטואליים אלא גם עבור מבני נתונים וקוד של מערכת ההפעלה. לעיתים יש צורך לפנות מסגרות. פינוי מסגרות אורך זמן כיוון שיש צורך לחפש דפים מאימים לפינוי וכן כיוון שדפים ששונו ביזרון צריכים להיכתב בחזרה למסגרת בדיסק לפני הפינוי. מצד שני חשוב שתבצע הקצאה מהירה של המסגרות. לכן, משתדלת מ"ה לשמור על מלאי של מסגרות פנויות ביזרון הפיזי. מדיניות זו מתבצעת ע"י 2 מנגנונים: הראשון מבוסס על שגרה של מ"ה המופעלת כל זמן קבוע ומנסה לפנות מסגרות. המנגנון השני הוא מנגנון חירום המופעל כאשר מלאי המסגרות קטן מדי. המנגנון הראשון מסתמך על רשימת מסגרות שצריך לכתוב לדיסק לפני מחיקתן (**מסגרות מלוכלכות**) ורשימה נוספת המכילה **מסגרות נקיות**. בנוסף, המנגנון כולל 3 שגרות:

השגרה **הראשונה** מוצאת מסגרות מיועדות לפינוי (ע"פ מדיניות מ"ה שתוסבר בהמשך) ומוסיפה אותן לאחת מ-2 הרשימות: מסגרות מלוכלכות או מסגרות נקיות (ע"פ מצב המסגרת). תפקיד השגרה הוא ליישם את מדיניות מ"ה בדבר פינוי מסגרות שאינו ישפיע על הביצועים.

השגרה **השנייה** "מנקה" מסגרות מרשימת המסגרות המלוכלכות ע"י כתיבתן לדיסק. מטרתה היא להגדיל את רשימת המסגרות "הנקיות" כיוון שאותן מהיר לפנות במקרה חירום. בנוסף, השגרה מנצלת זמן בו הדיסק אינו עסוק כדי לכתוב מסגרות אליו (וכך מונעת המתנה במצב בו צריך לכתוב את המסגרות כאשר הדיסק אינו פנוי).

השגרה **השלישית** עוברת על המסגרות הנקיות ומפנה אותן סופית (למשל מוחקת את התוכן). שגרה זו אינה חיונית כיוון ש-2 השגרות האחרות מבטיחות מלאי של מסגרות לפינוי מידי במקרה הצורך.

שגרות אלו מתעוררות מדי פעם ומנקות מספר מסגרות שונה בהתאם לכמות המסגרות הפנויות בהשוואה לכמות שמ"ה רואה כאופטימאלית (ייתכן והן לא יפעלו על אף מסגרת). הסיבה שמשתמשים ברשימת מסגרות "נקיות" ולא מנקים אותן סופית היא כיוון שבמקרה של חריג דף בו המסגרת נמצאת ברשימת המסגרות הנקיות, מ"ה תוציא את המסגרת מרשימת "הנקיות" ותספק את הבקשה ללא צורך בפניה לדיסק.

ניתן להוכיח כי המסגרות שכדאי ביותר לפנות הם אלו שהשימוש הבא בהן הוא הרחוק ביותר בזמן. סדר זה ממזער את מספר חריגי הדף. עם זאת, למ"ה אין מידע על שימוש עתידי אך היא יכולה להסיק מהעבר הקרוב. מדיניות **LRU** מפנה מסגרות לפי סדר השימוש האחרון בהם. ניתן להוכיח שמדיניות זו אופטימאלית במצב זה.

שימוש ב-LRU מצריך החתמה בחותמת זמן כל פעם שעושים שימוש בדף. אומנם ניתן לממש זאת אך ברוב המעבדים היכולת אינה קיימת. במקו, מעבדים רבים מדליקים סיבית (**Accessed 1x Used**) ברשומת המיפוי ב-TLB בכל פעם שהם משתמשים בדף. הסיבית מועתקת לטבלת הדפים בכל פעם שהמעבד מוחק מיפוי מה-TLB. מ"ה משתמשת בסיבית זו כדי לקרב את מדיניות LRU. הסיבית מאופסת כל פרק זמן קבוע. כאשר מ"ה מחפשת קורבנות היא נמנעת מלבחור מסגרות בהם הסיבית דולקת. ניתן לקרב את מדיניות LRU אף טוב יותר ע"י שמירת סיבית Used לפני איפוסה.

מערכות הפעלה רבות משתמשות במדיניות LRU או קירוב שלה בשני שלבים. בשלב הראשון נבחרים תהליכים שדפים שלהם יפונו ממסגרות פיזיות ובשלב השני מחליטה מערכת ההפעלה איזה מסגרות לפנות. לינוקס מפנה דפים של התהליך שגרם למספר הקן ביותר של חריגי דף לאחרונה. חלונות משתמש במנגנון מתוחכם יותר הנקרא **Working Sets**. הכוונה היא למספר המסגרות שהתהליך זקוק להן לפי הערכה של מ"ה. כאשר יש צורך

לפנות מסגרות, מערכת ההפעלה מפנה מסגרות באופן שבו כל תהליך יקבל את מספר המסגרות שהוא זכאי להן. בפרט, סביר שבעקבות חריג דף תפונה מסגרת של התהליך שגרם לחריג. מ"ה קובעת את ה-WS של תהליך לפי קצב חריגי הדף שהוא גורם. תהליך שגורם לחריגי דף רבים זקוק כנראה ליותר דפים ולכן ה-WS שלו יגדל ולהפך. גודל ה-WS מוגבל מלמטה ומלמעלה. הרף המינימאלי נועד לתגמל תהליכים שלא מבצעים מספר רב של חריגי דף והמקסימאלי נועד כדי להגן על המערכת כולה מפני תהליכים שגורמים למספר רב של חריגים.

למדיניות זו 2 יתרונות: הראשון הוא התייחסות הוגנת יותר של מ"ה (יותר הוגן לפנות מסגרת של תהליך המשתמש בהרבה מסגרות לעומת תהליך שמשמש במעט). השני הוא בכך שרוב מבני הנתונים של טבלאות דפים אינן מאפשרות התייחסות שווה לדפים של כל התהליכים באופן יעיל. בטבלאות שטוחות והיררכיות למשל, חותמת הזמן נשמרת בטבלאות דפים שכל אחת מייצגת מרחב זיכרון אחר. כדי למצוא מסגרת עם תאריך השימוש הרחוק ביותר, עליה לעבור על כל הטבלאות (כאשר רוב המידע אינו רלוונטי כלל כיוון שהוא ממופה לדיסק). לעומת זאת, אם קודם מתבצע בחירה של תהליך, מ"ה צריכה לעבור רק על טבלת הדפים שלו. בטבלת דפים הפוכה קל להתייחס באופן שווה לדפים של כל התהליכים ולכן אין לכך חשיבות.

אזורי דפדוף ומיפוי קבצים

מ"ה משתמשת ב-2 סוגים של אזורי דפדוף. הראשון משמש למסגרות אנונימיות (עבור מחסניות, משתנים גלובליים, מצבורים וכו'). אזור דפדוף זה עשוי להשתמש בדיסק שלם, מחיצה או קובץ רגיל (בלינוקס משתמשים במחיצות ובחלונות בקובץ רגיל ששמו c:/pagefile.sys). בלינוקס ניתן גם להגדיר אזורי דפדוף באופן דינאמי או להפעיל את המערכת ללא אזורי דפדוף כלל. בנוסף, ניתן לתת עדיפות לאזורי דפדוף (וכך להשתמש בדיסקים מהירים לשיפור ביצועים אך לאפשר אזור רחב יותר גם עם דיסקים רגילים).

הסוג השני משמש ל**מיפוי קבצים** למרחב הוירטואלי. על מנת להשתמש באזור דפדוף כזה, תוכניות (או מ"ה) מבקש בעזרת קראית מערכת להקצות דפים במרחב הזיכרון הוירטואלי שימופו לקובץ מסוים (או חלק ממנו). חריג דף על דף שממופה בצורה כזו גורם להבאת מידע מהקובץ למסגרת בזיכרון וכתיבה לדף גורמת לכתיבה לקובץ (כתיבה כזו קריאה לתהליכים אחרים מיד לאחר הכתיבה לזיכרון הממופה גם אם עוד לא נכתבו לדיסק). השימוש העיקרי במיפוי קבצים הוא לטעינת קוד לזיכרון. שיטה זו יעילה כיוון שאינה גורמת לטעינה של כל הקוד באופן מיידי לזיכרון הפיזי אלא רק במקרה של צורך אמיתי בקוד (בעקבות חריג דף). כך נחסך זמן טעינה מיותר לזיכרון הפיזי וכן נחסך המקום שפקודות שאינן בשימוש היו תופסות.

שיתוף זיכרון בין תהליכים ו-DLL

מנגנון הזיכרון הוירטואלי מאפשר להפריד בין מבני נתונים של תוכניות שונות ולהגן עליהם אך הוא גם מאפשר לשתף מבני נתונים בין תוכניות. כפי שצוין קודם, מבני הנתונים של מערכת ההפעלה והקוד שלה ממופים לכל מרחבי הזיכרון. ניתן אך גם למפות מבני נתונים וקטעי קוד שאינם של מערכת ההפעלה למספר מרחבים. אחת הסיבות לביצוע מיפוי כזה הוא במקרה בו מספר תוכניות משתמשות באותן השגרות. כך ניתן לחסוך מסגרות שאחרת היו מתבזבזות על עותקים של אותו מידע סטטי שאינו משתנה. שגרות אלו נבנות כך שכל **כתובת הינה יחסית למיקום תחילת השגרה** ולכן ניתן למקם אותה בכל מקום במרחב הוירטואלי. בזמן בניית התוכנית, במקום להוסיף עותק של הקוד לתוכנית, ה-Linker מוסיף סימון סימבולי לספרייה. בעת ריצת התוכנית, מ"ה מזהה סימון זה, ממפה את השגרות במרחב הוירטואלי ומוסיפה פקודות מכונה לקריאה לשגרות.

סיבה אחרת לשיתוף זיכרון הוא כדי להעביר מידע בין תהליכים. ניתן לבצע זאת גם על ידי קריאה וכתיבה מקבצים, אך מיפוי זיכרון היא הדרך היעילה ביותר. שיתוף זיכרון פוגע בהגנה של הפרדת תוכניות ושגיאה בתוכנית אחת עלולה לגרום גם לתוכניות אחרות לקרוס. עם זאת, החשיפה במקרה זה קטנה ביחס למצב בו כל תוכנית יכולה לגשת לזיכרון של כל תוכנית אחרת. בנוסף, ניתן להגן על תוכניות על ידי עידון ההרשאות למינימום הנדרש (למשל, תוכניות שקוראות בלבד לא יקבלו הרשאות כתיבה).

תזמון מעבדים ומיתוג תהליכים

תהליכים ומיתוג תהליכים

מערכת ההפעלה מספקת אבסטרקציות של מחשבים וירטואליים (תהליכים) ושל מעבדים (חוסים). כאשר אנו מריצים תהליכים רבים על מחשב עם מעבד אחד או מעט מעבדים, יש צורך למתג את המעבד בין כל התהליכים. לשם כך קיים מנגנון מיתוג ואלגוריתם המחליט איזה תהליך ירוץ בכל רגע נתון על כל מעבד.

תהליך (Process) – הוא מחשב וירטואלי. הוא מיוצג מ"ה על ידי מבנה נתונים שכולל את המרכיבים הבאים: **מצב המעבד** – על מנת לשחזר פעולת תהליך, יש לשמור את מצב המעבד ברגע שהוא הפסיק לפעול. לשם כך קיים מבנה נתונים הכולל את מצבו של המעבד (תוכן האוגרים, מצביע התוכנית, מצביע המחסנית וכו'). כאשר יש צורך להחזיר את התהליך לפעולה, מ"ה מחזירה את המעבד למצב השמור. **מפת הזיכרון** – ממפה את המרחב הוירטואלי של התהליך לזיכרון הפיזי ואזורי הדפדוף.

טבלת שגרות איתות – לתהליך יש מנגנון פסיקות וירטואלי הנקראים **איתותים (Signals)** באמצעותו ניתן להודיע לתהליך על אירועים שונים (סיום פעולת קלט/פלט, חריגים ועוד). תהליך יכול להגיב לאיתותים באמצעות קביעת שגרה ל טיפול בכל אחד מסוגי האיתותים. בזמן שתהליך אינו רץ, מ"ה צריכה לזכור כיצד התהליך רוצה לטפל בכל שגרה. בנוסף עליו לזכור האם התרחשו אירועים שלא טופלו עדיין (ולטפל בהם ברגע שהתהליך חוזר לפעולה).

זהות והרשאות – מ"ה זוכרת מי המשתמש שהריץ את התהליך (כי הרשאות התהליך נגזרות מכך). במ"ה מסוימות ניתן לשנות את הרשאות התהליך ביחס למשתמש שהריץ אותו ומ"ה צריכה לזכור גם שינויים אלו. **רשימת משאבים זמינים** – מ"ה מחזיקה רשימה של המשאבים שהתהליך ביקש (באמצעות "פתיחת" המשאב) וקיבל גישה אליהם (לאחר בדיקת הבקשה) וסוג הגישה שאושרה. במידה ובקשה למשאב אושרה, המשאב נוסף לרשימה ומוחזר **מזהה משאב (Handle/File Descriptor)** לתהליך המצביע לאיבר המתאים ברשימת המשאבים. כל רשומת משאב מכילה את סוג הגישה שאושרה ומצביע למבנה הנתונים שמתאר את המשאב במבני הנתונים של מ"ה. במקרים רבים רשימת המשאבים שמורה בטבלה ואז ניתן להשתמש במיקום בטבלה כמזהה המשאב. סגירת המשאב מוחקת אותו מהרשימה. לשימוש במנגנון זה 2 סיבות: חיסכון בבדיקת הרשאות בכל גישה ו**מניעת "זיוף" מצביע למשאב** (באמצעות הצבעה לא ישירה).

מצב תזמון וסטטיסטיקות שונות – מ"ה זוכרת, עבור תהליך שלא רץ, מדוע הוא לא רץ כדי לדעת האם ניתן להחזיר אותו לפעולה או שהוא מחכה לאירוע כלשהו. בנוסף נשמרים נתונים על הפעילות של התהליך בעבר. באמצעות מידע זה יכולה מ"ה לקבל החלטות לגבי תזמון התהליך וניהול הזיכרון הוירטואלי שלו (רצון להחזיר תהליך לריצה על המעבד האחרון עליו הוא רץ, קביעת גודל הקצאת זיכרון ע"פ כמות חריגי דף וכו').

חוט (Thread) – הוא מעבד וירטואלי (אחד מכמה) במחשב וירטואלי. לכן, כל החוטים באותו תהליך משתפים מפת זיכרון אחרת ובדרך"כ גם את ההרשאות, רשימת המשאבים ומצב האיתותים. מצב המעבד ומצב התזמון של החוט הינם הנתונים הפרטיים לכל חוט.

בלינוקס, ניתן להגדיר אילו מאפיינים משותפים לשני חוטים ואילו מאפיינים ייחודיים לכל חוט (למשל זיכרון משותף אך רשימת משאבים נפרדת). בחלונות ניתן לתת **לתהליך הרשאות נוספות**. במצבים כאלו מתערפל ההבדל בין תהליך לחוט.

מיתוג תהליכים

תהליך המיתוג נקרא **Context Switching**. לפני המיתוג, מעבד פיזי מסוים ממופה לחוט מסוים של תהליך כלשהו. הוא משתמש בפקודות המכונה של החוט ומשתמש במרחב הזיכרון של התהליך. ביצוע המיתוג יכול לקרות כתוצאה מבקשת שירות של מ"ה ע"י התוכנית או בעקבות פסיקה אחרת הגורמת להפעלת שגרת הפסיקה. כדי לבצע מיתוג בין תהליכים, תחילה עובר המעבד הפיזי לבצע שגרה של מערכת ההפעלה. התוכנית שרצה עד כה מושהה ומצב המעבד נשמר במחסנית. מ"ה מטפלת באירוע שגרם למעבר לשגרה של מ"ה (טיפול בבקשת השירות של התוכנית או טיפול בפסיקה החומרה). מ"ה מחליטה שתהליך או חוט אחר צריכים לרוץ על המעבד הפיזי (פירוט מדיניות בהמשך) והיא שומרת את מצב החוט או התהליך הקודם במבנה נתונים שמתאר אותו. אם הוחלף תהליך, מצביע טבלת הדפים מעודכן להצביע אל טבלה העדכנית. לסיום, משחזרת מ"ה את מצב המעבד של המעבד הוירטואלי שצריך לרוץ כעת ומתחילה להריץ אותו.

תהליכי וחוטי גרעין

תהליכי גרעין או חוטי גרעין הינם תהליכים מיוחדים בהם משתמשת מ"ה כדי לממש מנגנונים שונים הרצים כל זמן מה (למשל תהליך פינני מסגרות). תהליכים אלו משתמשים רק במבני הנתונים של מערכת ההפעלה. לכן, אין צורך להחליף את מרחב הזיכרון כאשר הם מופעלים (כי זיכרון מערכת ההפעלה ממופה לכל מרחבי הזיכרון). שימוש בתהליכי גרעין חוסך לפחות החלפת מפת זיכרון אחת (ואם התהליך הקודם ממשיך לפעול בתום תהליך הגרעין חוסכים החלפה נוספת). מ"ה משתמשת בתהליכים כאלו כאשר היא רוצה שהם יתחרו על זמן מעבד עם שאר התהליכים כדי לנצל זמן של חוסר פעילות (למשל כדי לכתוב מסגרות מלוכלכות לדיסק). במידה ותהליכים רגילים מוכנים לפעולה, תהליכי גרעין יקבלו פחות זמן מעבד. לעיתים משתמשים בתהליך כזה עם עדיפות נמוכה מאוד כדי "למלא את החלל" כאשר המעבד לא צריך לעשות דבר.

שדונים (Daemons) ושירותים (Services) הם תהליכים של מערכת ההפעלה אך אינם תהליכי גרעין (למשל מנגנון רישום האירועים או תהליכים שצריכים זיכרון וירטואלי פרטי מלבד הזיכרון של מ"ה).

יצירת תהליכים בלינוקס ויוניקס

בלינוקס ויוניקס ניתן ליצור תהליך חדש בצורה יעילה באמצעות קריאת **fork**. קריאה זו משכפלת למעשה תהליך קיים וכאשר היא חוזרת, היא חוזרת כשני תהליכים: המקורי והחדש. בכל אחד מהם, הקריאה מחזירה ערך שונה כדי שיהיה ניתן להבדיל ביניהם. פרט לכך, הם זהים לחלוטין. למרות ששני התהליכים זהים הם נפרדים. פעולות באחד מהתהליכים לא משפיע על התהליך השני. דרך אחת ליצור תהליך חדש הוא להעתיק את כל המסגרות (בדיסק ובזיכרון) של התהליך הקיים למסגרות פנויות. מצב זה אינו יעיל כיוון שיש להעתיק

מסגרות רבות וכיוון שלא ניתן להמשיך את התהליך המקורי עד שהעתקה מסתיימת. בנוסף, חלק גדול מהמסגרות שיועקו לא ישויות בשני התהליכים ועשוי להיווצר בזבוז זיכרון.

בלינקס ויוניקס משתמשים במנגנון **Copy On Write** בעת שכפול תהליכים. המנגנון מבוסס על מיפוי זיכרון משותף לשני התהליכים עד לנקודת הזמן בה אחד מהם משנה מסגרת. במקרה כזה מ"ה מעתיקה את המסגרת כך שלכל תהליך יהיה עותק פרטי. כדי להשתמש במנגנון, בעת פעולת fork, מ"ה מסמנת את כל המסגרות הממופות עם הרשאות קריאה בלבד ויוצרת עותק של טבלת הדפים עבור התהליך החדש. בטבלה כל הדפים מסומנים להתנהגות COW. כאשר אחד התהליכים ינסה לכתוב לזיכרון, ייוצר חריג דף בגלל שאין הרשאת כתיבה. בעת טיפול בחריג, מ"ה תזהה שהחריג נוצר כתוצאה מ-COW. היא תיצור עותק של המסגרת, תפנה את אחת מטבלאות הדפים למסגרת החדשה ותוריד את סיבית ה-COW ב-2 טבלאות הדפים שהצביעו למסגרת המקורית כי עכשיו המסגרות נפרדות (המנגנון למעשה מעט יותר מורכב כי יכול להיות שיש יותר מ-2 תהליכים המשתפים מסגרת COW וכיוון שיש להתחשב ב-COW גם כאשר מעבירים מסגרות בין הזיכרון לדיסק).

בחלונות הגישה שונה. במערכת זו ניתן ליצור תהליך חדש בלבד (התהליך היוצר מציין איזה תוכנית התהליך החדש יריץ). גישה זו זולה יותר כאשר רוצים להריץ תוכנית חדשה מאשר בלינקס ויוניקס אך יקרה יותר כאשר רוצים "לפצל" תהליך לצורך ביצוע פעולות קטנות יחסית באמצעות אותה התוכנית.

מדיניות לתזמון מעבדים

בכל פעם שמערכת ההפעלה שוקלת האם למתג מעבד, ישנה רשימה של חוטים הממתינים לרוץ וקבוצה משלימה של חוטים הממתינים לאירוע. מערכת ההפעלה צריכה להחליט למי להקצות את המעבד באמצעות מדיניות התזמון שלה. למדיניות התזמון שתי מטרות עיקריות העומדות לעיתים בסתירה:

שימוש יעיל במעבדים פזיים - מיתוג תהליכים צורך זמן מעבד ומטרת המדיניות היא למזער את אחוז של זמן המעבד שמתבזבז על מיתוג.

זמן תגובה נמוך - למשתמשים חשוב שתוכנית תגיב תוך זמן מהיר בהתאם לסוג הבקשה (מספר אלפיות שנייה עבור משאבים אינטראקטיביים אך אפילו תוך שעה עבור סימולציות למשל). נשים לב שאין צורך להקטין את זמן התגובה עד אינסוף (אין הבדל בין תגובה לעכבר תוך מיליונית שנייה לעומת אלפית שנייה). חשוב לציין שמחקרים הראו שאחידות בזמן התגובה חשובה למשתמשים יותר מזמן תגובה ממוצע קצר.

בנוסף ל-2 המטרות העיקריות, עשויות להיות קיימות מטרות נוספות למדיניות התזמון:

זמן תגובה מובטח לאירועים מסוימים - ישנם מערכות הדורשות תגובה לאירועים מסוימים תוך פרק זמן ידוע מראש. מ"ה המבטיחות זאת נקראות מערכות ז"א. יוניקס, לינקס וחלונות מורכבות מכדי להיות מערכות ז"א.

תזמון דביק - במחשבים מרובי מעבדים רצוי בדר"כ להריץ חוט או תהליך על אותו מעבד פזי בכל פעם שהוא נכנס לריצה. גישה כזו ממזערת את מספר חריגי המטמון וחריגי TLB.

תזמון כנופיות - במחשב מרובה מעבדים המריץ בעיקר תהליכים מרובי חוטים, רצוי בדר"כ להכניס לריצה את כל החוטים הלא ממתינים של תהליך למעבדים שונים (כדי למנוע המתנה תוך זמן קצר של חוט אחד לאחר).

תזמון מעבדים עם Multilevel Feedback Queues

Multilevel Feedback Queues - הינם מבני נתונים המאפשרים להגדיר טווח רחב של מדיניות תזמון ועליהם בדר"כ מבוססת מדיניות התזמון של מערכת ההפעלה. מבנה הנתונים מורכב ממערך של תורים. התור הגבוה ביותר (במקום 0) מכיל את התהליכים שנמצאים בעדיפות גבוהה ביותר לריצה. עדיפות שאר התורים יורדת על פי מיקומם. התור בו נמצא תהליך נקבע ע"י מערכת ההפעלה אך המשתמש יכול להשפיע על מיקומו (יפורט בהמשך). לכל תור קיים פרק זמן מקסימאלי שתהליך השייך אליו יכול לרוץ ברצף (לתור בעל העדיפות הגבוהה ביותר זמן הריצה הרצוף הנמוך ביותר). מבנה הנתונים מגדיר גם חוקים למעבר בין מתור לתור. מערכת ההפעלה בוחרת תמיד את התהליך שבראש התור הגבוה ביותר שאינו ריק. תהליך מפסיק לרוץ כאשר הוא רץ יותר מפרק הזמן שהוגדר לתור ויש עוד תהליכים בתור שלו, כאשר תור בעל עדיפות גבוהה יותר מתמלא או כאשר התהליך ממתין לאירוע. כאשר מפסיקים להריץ תהליך מבלי שהוא נכנס למצב המתנה לאירוע, מחזירים אותו לסוף התור ממנו הוא בא.

מדיניות מעבר מתור לתור - ישנם 3 סוגי חוקים המורים למערכת ההפעלה מתי להעביר תהליך מתור לתור:

1. תהליך שצבר בזמן האחרון זמן מעבד רב יועבר לתור נמוך יותר כדי למנוע הרעבה.
2. תהליך שממתין זמן רב למעבד יעבור לתור גבוה יותר (שוב כדי למנוע הרעבה).
3. תהליך שמחכה לאירוע גורם להכנסת התהליך לתור גבוה יותר בדר"כ (כיוון שבמקרים רבים כל עוד התהליך לא רץ לא ניתן לשחרר את המשאב אותו התהליך ביקש ומצב זה יכול לגרום לתהליכים אחרים להמתין).

אם המשתמשים היו יכולים לשלוט ישירות על העדיפויות, הם היו יכולים להגיע בקלות למצב של הרעבת תהליכים או ניצול לא יעיל של משאבים. במקום זאת, רוב מ"ה מאפשרות למשתמשים שליטה על הפרמטרים של חוקי המעבר.

ניתן להשתמש באחת מ-2 שיטות (או בשילוב של שתיהן) כדי להעריך את זמן המעבד שתהליך מסוים צרך כדי להחליט אם להעלות או להוריד אותו לתור אחר. השיטה הראשונה היא לזכור את הסיבה בגללה הפסיק לפעול ובמידה והופסק כיוון שמצאה את פרק הזמן, ניתן להוריד אותו תור (כתהליך שצרך זמן מעבד רב). דרך מורכבת יותר היא לשער את צריכת זמן המעבד של כל התהליכים כל פרק זמן ולהעבירם בין תורים. כדי לממש שיערוך זה, משתמשים בפונקציה מערכתית (כך שלזמן הריצה ב"סיבוב" הקודם חשיבות נמוכה יותר מזמן הריצה בסיבוב האחרון).

הענקת פרקי זמן קצרים לריצה בתורים גבוהים ופרקי זמן ארוכים לתורים נמוכים מממשת מדיניות של זמן תגובה מהיר לתהליכים אינטראקטיביים וניצולת מעבד גובהה כאשר אין תהליכים כאלו. תהליכים אטרקטיביים נוטים לצרוך מעט זמן מעבד ולהמתין הרבה לקלט מהמשתמש ולכן יהיו רוב הזמן בתורים גבוהים. כאשר אין תהליכים כאלו, תהליכים מתורים נמוכים ירוצו זמן רב יחסית ויגדילו את נצולת המעבדים.

חסרון שיטה זו היא שפעולתה מושפעת מפרמטרים רבים ולמרות שהם מקנים לו גמישות, הם מקשים על כיוונו.

תכנות תהליכים בו זמניים

ישנם מספר סיבות לממש תוכנית עם מספר תהליכים:

- רצון לנצל מספר מעבדים פיזיים.
- רצון לנצל את המעבד בזמן שהתוכנית מחכה להתקן איטי (תהליך אחד ממתין ואחר רץ בנתיים).
- מתן אשליה למשתמש שמספר פעולות קורות במקביל.
- מימוש שרתים בהם כל לקוח מטופל ע"י תהליך או חוט נפרד.

3 הסיבות האחרונות מייצגות שאיפה למודולריזציה טמפורלית – הרצון לממש תוכנית בצורה מודולארית תוך שימוש בממשקים ברורים בין מודלים וללא תלות נוספת ביניהם. כאשר 2 מודלים צריכים לפעול במקביל, אין דרך נוחה ליישם זאת בתכנות סדרתי. לשם כך, צריך למתג בין המודלים באופן מפורש. זו פגיעה במודולאריות. פותרים זאת באמצעות שימוש בחוט לכל מודל. בנוסף, גם במ"ה ישנם פעולות המבוצעות ב"ז". למשל, במקרה של פסיקה, שגרת הפסיקה והתהליך שהופסק הם כמו 2 תהליכים הרצים בו"ז. במ"ה למעבדים מרובי לבות יש צורך להשתמש במנגנוני תכנות תהליכים בו"ז (בנוסף לפסיקות) על מנת להשיג נכונות ויעילות.

חוטים (והשירותים שלהם) פותחו לאחר שהתבררו בעיות במנגנונים דומים וישנים יותר. למשל, שימוש בתהליכים רגילים גרם למחיר גבוה למיתוג בין תהליכים ושימוש בשירותים (כמו סמפורים) היה קשה יותר.

מנגנוני החוטים

שימוש בחוטים מתבצע באמצעות 3 מנגנונים:

מנגנון ליצירת חוט חדש – יצירה של חוט חדש מתבצעת בעזרת קריאת מערכת המריצה שגרה אך לא ממתנה לסיום פעולתה. במקום זאת, 2 השגרות (הקוראת והנקראת) רצות בו"ז.

מנגנון למניעה הדדית – קריאה רגילה לשגרה (לא בחוט נפרד) מהווה מעין תיאום בין השגרה הקוראת לנקראת (כיוון שכאשר הקוראת ממשיכה לרוץ היא יודעת שהשגרה הנקראת בוצעה). כאשר מריצים מספר חוטים בו"ז דרוש לעיתים תיאום ביניהם. צורת התיאום הפשוטה ביותר מבוססת על **מנעול (Mutex או Lock)**. ניתן לדמות מנעול לקופסה שיכולה להכיל מזהה של חוט יחיד. כל עוד קיים מזהה בה המנעול אינו זמין ("נעול"). כאשר היא ריקה, המנעול זמין ("פתוח"). חוט יכול לנסות לנעול מנעול. כאשר שגרת הנעילה חוזרת, החוט יודע שעתה המנעול בבעלותו. השימוש במנעולים נועד להבטיח **אטומיות** (כלומר, הבטחה שבעת ביצוע סדרת פעולות מסוימת על מבנה נתונים אף חוט אחר לא ניגש למבנה עד שהסדרה מסתיימת).

אירועים (או משתני תנאי) – הם עצמים המאפשרים לחוט לחכות לאירוע או להודיע לחוטים אחרים שאירוע מסוים קרה כבר. שימוש במנעולים במקרים כאלו מיישם למעשה שיטה של דגימה (וזו כאמור אינה יעילה) ולכן נוצרו אירועים. חוט הממתין לאירוע מסוים (למשל חוט קורא הממתין למילוי של חוצץ), יכנס למצב המתנה עד קיום האירוע. חוט אחר יוכל בינתיים לבצע פעולות כדי לגרום לאירוע להתרחש (למשל חוט כותב ימלא את החוצץ ויכריז על קיום האירוע). החוט שהכריז על האירוע לא צריך לדעת על קיומם של החוטים האחרים. הוא אך ורק מודיע למערכת ההפעלה על קיום האירוע והיא זו שדואגת להעיר את החוטים הממתנים (אם יש כאלו).

ביוניקס ולינוקס, ע"מ להמתין לאירוע c יש להשתמש בפקודה pthread_cond_wait(&c, &m). פקודה זו משחררת את מנעול m ונכנסת להתנה (פעולה אטומית). לא ייתכן שבין שחרור m והכניסה להתנה יתרחש אירוע c). כאשר אירוע c מתרחש, החוט משחרר מהמתנה ומיד נועל את m. כדי להודיע שאירוע מסוים אירע משתמשים ב-pthread_cond_signal(&c) (משחררת את אחד החוטים הממתנים לאירוע - אין סדר המתנה כלשהו) ו-pthread_cond_broadcast(&c) (משחררת את כל החוטים הממתנים לאירוע). חשוב לזכור ש-2 הפונקציות משחררות רק חוטים שכבר ממתנים. כלומר, אירוע הוא רגעי וחולף ואינו מצב בינארי שהמערכת זוכרת.

חיות וקיפאון

קיפאון (Deadlock) – הוא מצב שבו כל אחד מהחוסים מחכה שחוט אחר ישחרר מנעול או יכריז על אירוע (כל החוסים נמצאים במצב המתנה לעד) והוא נגרם כתוצאה משימוש לא נכון במנגנוני התיאום. זו תקלה שקל באופן יחסי לאבחון באמצעות דיבגר (כיוון שבדרך"כ כל החוסים נתקעים וניתן לבדוק למה). קיפאון יכול להיגרם למשל כאשר חוט א' נועל את מנעול מס' 1. לאחר מכן, חוט ב' נועל את מנעול 2 ומנסה לנעול את מנעול 1. בינתיים מנסה חוט א' גם הוא לנעול את מנעול 2. שני החוסים יישארו תקועים כך לעד. נשים לב שקיפאון יכול להיגרם גם באמצעות שימוש לא נכון באירועים אך בעיה זו קלה יותר לפיתרון (ע"י שימוש בפקודת broadcast).

בצורה פורמאלית, ניתן לתאר מצב של קיפאון כגרף דו"צ עם קבוצת מנעולים וקבוצת חוסים. נסמן קשת מחוט למנעול אם החוט ממתין למנעול וממנעול לחוט אם המנעול נעול ע"י החוט. אם אין חוסים הממתינים לאירועים, קיים מצב של קיפאון אמ"מ קיים בגרף מעגל מכוון. במערכות כמו מסדי נתונים, מנגנוני התיאום כוללים מנעולים בלבד. לכן, קל להם לאתר מצב של קיפאון. במקרה כזה הם משחררים את אחד החוסים (לאחר שביטלו את הפעולות שביצע עד ההמתנה כדי לשמור על תקינות נתונים). מ"ה בדר"כ אינן מבחינות במצב של קיפאון כיוון שגילוי המצב אינו פשוט (כי במ"ה יש גם אירועים לצורך תיאום). כמו כן, בדר"כ רוב החוסים ממתינים במצב קיפאון ולכן המשתמש יכול לזהות את הבעיה ולהגיב. ובנוסף, בתוכניות רגילות, קשה לבטל פעולות שעשה חוט שנתקע. מצב קיפאון אינו ייחודי לחוסים ויכול להתקיים בכל מערכת המריצה תוכניות בו"ז.

ישנם 2 דרכים עיקריות כדי למנוע קיפאון בעקבות שימוש במנעולים והמתנה למשאבים. ע"פ הדרך הראשונה, לכל חוט יתאפשר לנעול רק מנעול אחד בו זמנית. במצב כזה, אם חוט נועל מנעול, לא ייתכן שהוא ממתין למנעול אחר ולכן לא ייוצר מעגל מכוון בגרף. ע"פ הדרך השנייה, יש לסדר את המנעולים בסדר כלשהו ולאפשר לכל חוט לנעול מנעולים רק לפי סדר זה. במצב כזה, גם לא יתאפשר מעגל מכוון (אלא אם חוט כלשהו הפר את הכלל). עם זאת, לא תמיד ניתן לסדר את המנעולים בסדר מסוים (למשל בפרוטוקול רשת, מידע מתוכנית לרשת עובר שכבות תקשורת בסדר הפוך מזה שעובר מידע מהרשת לתוכנית). במקרים כאלו צריך להשתמש באלגוריתמים מסובכים יותר כדי להבטיח חיות.

כדי למנוע מצב של קיפאון כתוצאה מהמתנה לאירועים, ניתן להגביל את זמן ההמתנה לאירוע (תחת ההנחה שהתוכנית תדע כיצד להגיב במצב של timeout).

חיות (Liveness) – מתייחס לתוכניות שאינן יכולות להיכנס למצב קיפאון.

גרעיניות

אנו משתמשים בחוסים על מנת לבצע פעולות שונות בו זמנית (או לפחות ללא שאחת תמתין לשנייה) אך אנו משתמשים במנעולים ואירועים על מנת לעכב פעולות עד שתנאים מסוימים יתממשו. הסיבה לכך הינה שלא כל סדר הפעולות האפשרי הוא נכון. עם זאת, בעת שימוש במנעולים, סדר הפעולות שאנו מאפשרים הינו, לעיתים, קטן מקבוצת סדר הפעולות הנכונות. כלומר, ישנם מקרים בהם שימוש עדין יותר במנעולים יכול לאפשר ביצוע יעיל ומהיר יותר של התוכנית.

גרעיניות גישה גסה זהו מצב בו מעט מנעולים מגנים על מבנה נתונים. לעומת זאת, **גרעיניות גישה עדינה** היא מצב בו מנעולים רבים מאפשרים לחוסים רבים גישה בו"ז לחלקים קטנים של מבנה הנתונים. גרעיניות עדינה מאפשרת לחוסים רבים גישה בו"ז למבנה הנתונים אך לצידה קיימת עלות של נעילת מנעולים רבים יותר מאשר גרעיניות גסה (צורך משאבים). לכן, ייתכן שכתוצאה מעידון הגרעיניות דווקא נגדיל את זמן הריצה של תוכנית. פתרון אידיאלי הינו הורדת המחיר של פעולות על מנעולים (ובעיקר נעילה כאשר המנעול פתוח ושחרור כשאין חוסים ממתינים). עם זאת, במ"ה רבות נעילה ושחרור אינן זולות ולכן מידת הגרעיניות תלויה (בנוסף לכמות המעבדים, משך הזמן שהחוט מבלה בטיפול במבנה הנתונים וכו') גם בעלות פעולות על מנעולים.

יעילות

למרות שמיתוג בין חוסים זול ממיתוג בין תהליכים, גם מיתוג חוסים היא פעולה יקרה. לכן, אנו מעוניינים למנוע, אם ניתן, מיתוג תכופ בין חוסים.

מיתוג מיותר עלול לקראת כאשר חוט מתעורר מהמתנה רק כדי לגלות שאינו יכול להתקדם ואז חוזר להמתנה. למשל, מצב בו אנו מודיעים על אירוע ב-broadcast כאשר בפועל רק חוט אחד יכול להתקדם (כל שאר החוסים "מתעוררים" לחינם וחוזרים להמתין). כדי לפתור בעיה זאת, ניתן להפריד אירועים לאירועים ספציפיים יותר ולהשתמש בקריאת single (בתנאי שבטוחים שהחוט שיתעורר יוכל לפעול).

מיתוג תכופ ומיותר נוסף יכול להיווצר כאשר מספר חוסים רב שאינו ממתין לאירועים או מנעולים גדול ממספר המעבדים הפיזיים. במצב כזה מ"ה תבצע מיתוג בין חוסים כדי לאפשר לכולם לרוץ באופן שווה. בדר"כ רצוי שמספר החוסים שאינם ממתינים לאירועים או מנעולים יהיה קרוב למספר המעבדים. זה פוגע במודולאריות אך עשוי לשפר ביצועים.

הגינות והרעבה

מערכת ההפעלה אינה מבטיחה הגינות בכל הנוגע למנעולים ואירועים. אחת הסיבות הינה כיוון שבמחשב מרובה ליבות, ייתכן ו-2 חוטים/תהליכים יבקשו את אותו מנעול בדיוק באותו הזמן. סיבה נוספת היא כיוון שפעמים רבות, מנגנון המנעולים ממומש בחומרה ואינו מבטיח הגינות. עם זאת, מ"ה כן מבטיחה בדרך כלל מניעת הרעבה. נשים לב כי גם אם מערכת ההפעלה הייתה מבטיחה הגינות, תוכניות לא היו מתנהגות בהכרח באופן הוגן ואף יכולה ליצור הרעבה (במקרה של תכנות לא נכון). אם אנו מעוניינים בהגנות (ואנו תמיד רוצים לפחות מניעת הרעבה), אנו צריכים לגבש מדיניות ולוודא שהתוכנית מממשת אותה באופן נכון.

עדיפות והיפוך עדיפויות

מ"ה מסוימות ניתן להגדיר עדיפויות לחוטים. עדיפות יכולה לתת לחוט יותר זמן מעבד או קדימות בקבלת מנעולים. במקרה הרגיל (פוסיקס וג'אוה) חוט בעל עדיפות נמוכה לא ירוץ כלל כל עוד חוט בעל עדיפות גבוה יכול לרוץ. מימוש זה מאפשר לתת קדימות לפעולות דחופות אך יישום פשטני של מנגנון העדיפויות עשוי להביא למצב לא רצוי הקרוי **היפוך עדיפויות**. מצב זה קורה כאשר חוט בעל עדיפות בינונית מונע מחוט בעל עדיפות נמוכה לרוץ ולכן החוט בעל העדיפות הנמוכה לא משחרר משאב שחוט בעל עדיפות גבוהה צריך. כך נוצר מצב בו חוט עם עדיפות בינונית רץ במקום חוץ עם עדיפות גבוהה. כדי לפתור בעיה זו, יש להעלות זמנית את העדיפות של החוט המחזיק במנעול, לעדיפות של החוט הזקוק למנעול זה. שינוי זה יגרום לשחרור מהיר של המנעול וכניסה לריצה של החוט המחכה. לא ניתן להפעיל אותה שיטה על אירועים כיוון שלא ניתן לדעת האם ומתי חוט יודיע על אירוע.

נושאים נוספים

Posix Threads – לממשק זה שירותים נוספים: מנעולי קריאה/כתיבה (rwlock) המאפשרים לכותב אחד או למספר קוראים לגשת למבנה נתונים, המתנה לסיום ריצת חוט (join) והפסקת ריצת חוט (cancel), משתנים לא משתופים (keys) ושליטה בעדיפויות ובמדיניות תזמון המעבדים. בנוסף, חשוב לשים לב כי קריאות מערכת ביוניקס נכתבו לפני המצאת החוטים ולכן כתיבתם הניחה כי רק קריאת מערכת אחת יכולה להתבצע בו"ז. למרות שסמנטיקה זו עודכנה, יש צורך בתשומת לב להתנהגות קריאות המערכת בתוכנית מרובת חוטים.

חוסים ב-Win32 – למרות שמנגנוני התיאום בחלונות דומה למימוש שתואר קודם, קיימים מספר הבדלים בסמנטיקה של אירועים. בחלונות, אירוע הוא מצב בינארי שזוכר שמאורע אירע בעבר. בנוסף קריאת המערכת שממתינה לאירוע אינה משתחרר מנעול. האירועים בחלונות מתחלקים ל-2 סוגים: **איפוס ידני ואוטומטי**. באיפוס ידני, מדליקים ומכבים את האירוע ידנית ע"י קריאת מערכת. הדלקת האירוע משחררת מנעילה את כל החוסים הממתינים. אירועים שיכנסו להמתנה בזמן שהאירוע דולק לא ימתינו כלל. באיפוס אוטומטי, מדליקים את האירוע באופן ידני אך הוא כבה אוטומטית מייד שהוא משחרר חוט מהמתנה. בנוסף, אם לא קיים אף חוט הממתין לאירוע, האירוע יישאר דולק עד שחוט יבצע עליו המתנה (יש קריאה שמעירה רק חוסים בהמתנה אך היא לא שימושית במיוחד). הבדל נוסף, הוא שבחלונות ניתן לתת לחוסים, אירועים ומנעולים שמות ולגשת אליהם מתהליכים אחרים. זה מייקר את השימוש בהם אך הופך אותם ליותר גמישים. כדי לפתור את בעיית היעילות, חלונות תומכת בסוג מנעולים נוסף הנקרא **קטע קריטי (Critical Section)**. מנעולים אלו ניתנים לשימוש רק בתהליך אחד והוא זול יותר מאשר שימוש במנעולים רגילים.

מערכות קבצים

המונח מערכות קבצים מתייחס לשני מושגים שונים: המושג הראשון אליו מתייחס המונח הינה לצורת ארגון של קבצים בדיסק ולשגרות שמטפלות במבנה הנתונים שמייצג את הקבצים. רוב מ"ה כיום תומכות במספר צורות ארגון קבצים (בין היתר, כולן תומכות למשל בתקן ארגון קבצים בתקליטורים). **המונח משמש גם להתייחסות למבנה נתונים ספציפי של קבצים** (בד"כ על דיסק או מחיצה) ולמרחב השמות שמתייחסים לקבצים הללו.

קובץ – מתייחס בד"כ לרצף של נתונים (מערך של בתים שגודלו לא נקבע מראש). ניתן לקרוא ולכתוב נתונים מכל מקום ברצף (קריאה לאחר סוף הרצף מחזיר בד"כ שגיאה וכתיבה אחרי הסוף מאריכה אותו). מ"ה מסוימות לא מאפשרות ליצור "חורים" בקובץ (ובמקרה זה ממלאות אותם בזבל או אפסים). אחרות, מאפשרות ליצור "חורים" אשר אינם תופסים מקום בדיסק (קריאה מ"חור" תחזיר שגיאה, אפסים או זבל).

מ"ה מסוימות משתמשות במונח קובץ כדי להתייחס לעצמים מורכבים יותר, בד"כ מספר רצפים. מקינטוש תומכת ב-2 תתי רצפים הנקראים התפצלויות (forks). הראשונה נקראת התפצלות הנתונים (data fork) והשנייה נקראת התפצלות המשאבים (resource fork). חלונות NT תומכת במספר בלתי מוגבל של רצפים. לראשון אין שם ומתייחסים אליו בעזרת שם הקובץ. לאחרים יש שמות וניתן להתייחס אליהם באמצעות שרשרת שמם לשם הקובץ (מופרד עם נקודתיים). זהו למעשה עידון של מרחב השמות (קובץ עם רצפים מקביל באנלוגיה לספריה עם מספר קבצים). יתרון ברצפים הינו שפעולות על הקובץ משפיעות על כל הרצפים. יתרון אחר הוא הוספת רצפים ללא שתוכניות אחרות צריכות להיות מודעות לכך (למשל מידע של תוכנית גיבוי). החיסרון העיקרי של תתי רצפים הינה בכך שאינם נתמכים בכל סוגי מערכות הקבצים. זה מקשה על העברת תוכניות שמשמשות ברצפים בין מ"ה והעברת קבצים עם רצפים למ"ה שאינן תומכות (ההתייחסות שלנו לקבצים תהיה כרצף בודד).

מרחב השמות

מ"ה תומכות במרחב שמות היררכי מבוסס על מחרוזות להתייחסות לעצמים. מרחב השמות הוא גרף מכוון ללא מעגלים עם שורש (במ"ה מסוימות מרחב השמות הוא פשוט עץ). על מנת להתייחס לעצם ע"י ציון שמו, צריך לתאר מסלול בגרף שמתחיל שורש ומסתיים בעצם הרצוי. בלינקס ויוניקס לעצמים עצמם אין כלל שמות ומציינים מסלול ע"י שרשרת שמות הקשות שמרכיבות אוו בין תווי הפרדה. בחלונות, לעצמים יש שם (או מספר שמות) וכדי לציין מסלול משרשרים את שמות הצמתים בין תווי הפרדה.

במרחב השמות בלינקס ויוניקס יכולים להופיע מדריכים (המכילים מצביעים לעצמים אחרים במרחב), מצביעים, קבצים, קבצים מיוחדים (המייצגים משאב), מצביעים סימבולים (יפורט בהמשך) וערוצי קשור משני סוגים (צינורות ושקעים). חלונות תומכת גם בסוגים נוספים של עצמים במרחב השמות, כמו מנעולים ואירועים.

הממשק של מערכת ההפעלה מאפשר לתוכניות לבצע פעולות על מרחב השמות. בין הפעולות האפשריות בלינקס ויוניקס (בחלונות ניתן לבצע מגוון דומה): יצירת עצם חדש ומצביע אליו ממדריך כלשהו (create, open) לקבצים רגילים, mknod לקבצים רגילים, התקני חומרה וצינורות, socket/bind לשקעים), יצירת מדריך חדש ומצביע אליו (mkdir), יצירת מצביע לעצם קיים (link), יצירת מצביע סימבולי (symlink), שינוי שם מצביע (rename), מחיקת מצביע (unlink). העצם עצמו נמחק כשאין מסלול מהשורש אליו), שינוי הרשאות גישה לעצם (chown, chmod ואחרים), **שתיילת מערכת קבצים שלמה במקום מסוים במרחב הקיים (mount) או הסרתה (unmount)**, פענוח שם למזהה פנימי והודעה למ"ה על השימוש בו כדי שלא ימחק (open) והודעה על סיום השימוש בעצם (close).

פענוח שמות – קריאת המערכת open מקבלת מסלול בגרף ומחזירה מזהה בעזרתו ניתן יהיה להתייחס לעצם בקריאות מערכת בהמשך. המסלול מצוין על ידי מחרוזת שמרכיבה מופרדים באמצעות / או \. מ"ה מפענחת את המסלול למזהה פנימי של העצם ע"י חיפוש כל מרכיב של השם במדריך שתחילית השם מציינת. התחילית הריקה מציינת תמיד את שורש מרחב השמות. כל מדריך בלינקס ויוניקס מכיל 2 מצביעים מיוחדים עם שמות קבועים: ". מצביע למדריך עצמו ו-". מצביע למדריך ההורה. עבור כל תהליך, מערכת ההפעלה מחזיקה מצביע למדריך אחד שנקרא **המדריך הנוכחי**. שמות שאינם מתחילים בתו הפרדה מפוענחים החל ממנו. מסלולים המתחילים מהמדריך הנוכחי נקראים **מסלולים יחסיים** בניגוד ל**מסלולים מוחלטים** שמתחילים מהשורש. פענוח השמות הוא תהליך הדורש זמן מעבד. פענוח שמות עם מסלול קצר יעיל יותר מפענוח שמות ארוכים (בפרט פענוח שמות יחסיים מהיר יותר בדרכ"כ משמות מוחלטים).

מצביעים סימבוליים – מצביעים סימבוליים הם קבצים מיוחדים שמכילים שם קובץ. אם בזמן פענוח שם מתברר למע"ה שתחילית של השם מתייחסת למצביע סימבולי, היא מתחילה מחדש את תהליך הפענוח כאשר התחילית מוחלפת בתוכן המצביע הסימבולי. מעבר לכך, היא אינה עושה שום פעולה עם מצביעים סימבולים (לא מודאת שהעצם המוצב קיים כאשר יוצרים מצביע סימבולי, מאפשרת למחוק עצם שיש אליו מצביע סימבולי וכו'). מצביעים אלו בדרכ"כ מאטים מעט את תהליך הפענוח כיוון שהם מוסיפים שלבים במהלכו.

נקודות הצבה (Mount Points) או נקודות פענוח (Reparse Points) – במערכת יוניקס, **קיימת טבלה של צמתים במרחב השמות** המצביעים על שורשים של מערכות קבצים שלמום. כאשר תהליך הפענוח מגיע לצומת כזה, הפענוח ממשיך מהשורש של מערכת הקבצים שמוצבת בנקודה זו. נקודת ההצבה חייבת להיות מדריך קיים. מערכת הקבצים של חלונות תומכת במנגנון דומה (נקודות פענוח). אלה הם צמתים במערכת הקבצים שמפנים את תהליך הפענוח לשורש של מערכת קבצים אחרת, למדריך במערכת קבצים אחרת או לשגרה כלשהי שתמשיך את תהליך הפענוח. הראשונה פועלת כמו ביוניקס, השנייה פועלת בצורה דומה אך מאפשרת עידון של מרחב השמות והשלישית מאפשרת לממש מערכות אחסון היררכיות. מערכות כאלו מזהות אוטומטית קבצים שלא נישאו אליהם זמן רב ומעבירות קבצים אלו למדיה זולה ואיטית יותר. לאחר העתקת הקובץ למדיה האיטית היא כותבת במקומו במערכת הקבצים נקודת פענוח הגורמת להפעלת שגרה של מערכת האחסון כשמנסים לגשת לקובץ (במקרה גישה מערכת האחסון תאתר את הקובץ בדיסק האיטי ותחזיר אותו). כאשר משתמש במערכות אחסון כאלו חשוב להימנע מלהריץ תוכניות גיבוי הרצות על כל הקבצים (אלא אם הן נמנעות מלגשת לקבצים דרך נקודות פענוח). נקודות הצבה משמשות "תפירת" מספר מערכות קבצים למרחב שמות אחד.

מעגלים ומחיקת קבצים – כיוון שמרחב השמות אינו עץ, יתכנו באופן עקרוני מעגלים בגרף. כיוון שיתכנו מספר הצבעות על קובץ, הקובץ אינו צריך להימחק כאשר מוחקים את אחת ההצבעות אלא כאשר אין יותר מסלול מהשורש לקובץ. מעגלים מקשים על הבחנה בין קבצים שניתן להגיע אליהם לבין קבצים שצריך למחוק (מכונים "זבל"). מ"ה מזהות זבל באמצעות ספירת מספר ההצבעות אל כל קובץ (כאשר הוא יורד ל-0 הקובץ נמחק). אם מתירים למעגלים להיווצר בגרף, קבצים עלולים להפוך ללא נגישים בלי שמספר ההצבעות ירד ל-0. אי לכך, מ"ה אינן מתירות בדרכ"כ ליצור מעגלים. הדרך הקלה והנפוצה ביותר לאכופף זאת היא **לאסור יצירת יותר ממצביע אחד למדריך**. אם יש מצביע אחד בדיוק לכל מדריך, תת הגרף המושרה מהמדריכים הוא עץ עם שורש ואין בו מעגלים. כיוון שדרגת היציאה של שאר הצמתים היא 0, לא יתכנו מעגלים במרחב השמות. אם מתירים ליצור מעגלים, ניתן למצוא את כל הקבצים שאינם נגישים מהשורש ולמחוק אותם באמצעות **איסוף זבל** (סריקה של כל מרחב השמות, סימון הקבצים הנגישים ומחיקת כל הקבצים שלא סומנו). שיטה זו אינה נמצאת בשימוש בדרכ"כ במ"ה. קיום מצביע סימבולי לקובץ אנו מונע את מחיקתו ולכן הביעה שיוצרים מעגלים לא נוצרת במעגלים עם לפחות מצביע סימבולי אחד (ולכן ניתן ליצור איתם מעגלים). תוכניות שסורקות את מרחב השמות צריכות להיות מודעות לכך (כדי לא להיכנס ללולאות אינסופיות או לסרוק את אותו הקובץ יותר מפעם אחת).

שמירת ואחזור קבצים

סמנטיקה של גישה לקבצים – אוסף חוקי התנהגות של גישה בו זמנית לקבצים. חוקים אלו נדרשים כיוון שקובץ כלשהו נגיש ליותר מתהליך אחד בו זמנית ומהווה למעשה זיכרון משותף למספר תהליכים. בנוסף, קבצים בדר"כ שמורים על דיסקים שאינם נדיפים ולכן, לעיתים, תהליך צריך לדעת האם נתונים שהוא כתב לקובץ נכתבו כבר לדיסק. כלומר, החוקים נועדו כדי לשמור על קונסיסטנטיות ועמידות.

קונסיסטנטיות – רוב מ"ה נוקטות בגישה לפיה כתיבות הן אטומיות. כלומר, אם תהליך כותב לקובץ אותו קורא תהליך אחר, התהליך הקורא יקרא את הנתונים לפני או אחרי הכתיבה אך לעולם לא תערובת של שניהם. בנוסף, כתיבה לקובץ ניראת מייד לתהליכים אחרים (כלומר כל בקשה לקריאה שהגיעה אחרי הכתיבה, תראה את המידע החדש). גישות לקבצים שאינם שמורים על דיסקים באותו מחשב, לא מקיימות בדר"כ תנאי זה.

עמידות – אירוע של נפילת מחשב ו עלייתו מחדש עלולה לגרום לביטול פעולות שלכאורה הסתיימו (כיוון שהמידע עדיין לא הועבר מהזיכרון הנדיף לזיכרון הלא-נדיף). מערכות הפעלה בדר"כ אינן מבטיחות עמידות כזו. עם זאת, ישנן מ"ה המבטיחות עמידות לאחר שהקובץ נסגר (אם לא הוחזרה שגיאה). במערכות יוניקס ניתן לעיתים לדרוש עמידות אחרי כל כתיבה באמצעות פתיחת הקובץ עם הדגל `0_SYNC`. כמעט תמיד, הבטחת עמידות פוגעת בביצועי מערכת הקבצים.

מיפוי קבצים

קבצים שמורים בבלוקים שגודלם קבוע. בלוקים השייכים לקובץ מסוים אינם בהכרח רצופים. אחרת, לא ניתן היה להגדיל קובץ כאשר הבלוקים העוקבים לו בשימוש קובץ אחר (אלא אם היינו מעבירים את הקובץ למקום אחר בדיסק – יקר). הבעיה השנייה הייתה היווצרות **פיצול היצוני** (כלומר היווצרות חורים בין קבצים במקום בו שכנו קבצים שנמחקו והיו קטנים מדי עבור קבצים חדשים שאנו רוצים ליצור). נשים לב שגודל הכולל של החורים יכול להיות מספיק גדול.

בלוקים גדולים משפרים את זמן הגישה הממוצע לנתונים וחוסכים מקום בטבלת הדפים (כי יש פחות בלוקים). עם זאת החיסרון שלהם הוא בבזבוז מקום הנקרא **פיצול פנימי** (כיוון שהבלוק האחרון בדר"כ לא מלא). מ"ה מסוגלות להשתמש בבלוקים בטווח גדלים מסוים כל עוד כל הקבצים משתמשים באותו גודל בלוק. היא בדר"כ תבחר בלוקים גדולים עבור דיסקים גדולים וקטנים עבור דיסקים קטנים. לעיתים מומלץ לבחור לבד את גודל הבלוקים (למשל אם אנו יודעים שמספר הקבצים קטן עדיף לבחור בלוקים גדולים כדי לשפר ביצועים).

במערכות קבצים מסוימות ניתן לחלק בלוק לשברים (fragment) עבור החלק האחרון של הקובץ. ב-FFS ניתן לפצל בלוק ל-4 או 8 שברים כאלו. השברים מקטינים מאוד את הפיצול הפנימי אך מסבכים כמובן את מבנה הנתונים. עם זאת, קיים פיצוי על כך בעובדה שהקובץ מורכב מבלוקים גדולים (פרט לאחרון).

ביוניקס ולינוקס מיפוי רצף הנתונים בקובץ לבלוקים מתבצע באמצעות **inode**. קובץ מיוצג ע"י inode שמחזיק (מלבד הרשאות גישה וזמן יצירה) את אורך הקובץ ומערך של אינדקסים לבלוקים בדיסק הממפה את רצף הנתונים לבלוקים. אולם מערך האינדקסים מוגבל בגודלו וכדי לאפשר למפות קבצים גדולים, משתמשים במיפוי היררכי. ב-inode שומרים מערך קטן של n פוינטרים ובסופו מצביע לבלוק עם פוינטרים לבלוקים הבאים (indirect block), אחריו מצביע לבלוק בגודל b עם מצביעים ל-indirect (הנקרא double indirect) ואחריו בלוק עם מצביעים ל-double indirect. סה"כ מספר הבלוקים המקסימאלי בקובץ הוא $n + b + b^2 + b^3$. גישה זו מאפשרת למפות קבצים גדולים מאוד (אם כי מוגבלים בגודלם) ובו בעת למפות קבצים קטנים ביעילות. inode של קבצים פתוחים נשמר בזיכרון ולכן אינו דורש גישה לדיסק עבור קבצים קטנים (מלבד הפעם הראשונה כמובן).

ברוב מערכות יוניקס ולינוקס, inodes נשמרים בנפרד מבלוקים של נתונים (לעיתים כולם שמורים באזור אחד של הדיסק ולעיתים בכמה). הקצאה כזו של inodes מביאה מספר inodes בכל גישה לאזור זה של הדיסק ולכן מאיצה גישה אליהם (ולפיכך גם פתיחת קבצים וסריקת מדריכים). האצה נוספת קוראת כיוון שבצורה כזו קל למצוא אותם. מצד שני, הצורך להקצות מקום מיוחד ל-inodes עשוי לבזבז מקום על הדיסק (אם יש מקום לנתונים אבל לא ל-inodes (ואז אי אפשר ליצור קבצים) או להפך). מערכות מתקדמות מקצות מקום לפי הצורך.

בחלונות **רשומות (records)** הן המקבילות ל-inodes ובאמצעותן מיוצג קובץ. כל הרשומות נשמרות בקובץ מיוחד שנקרא MFT. זהו קובץ רגיל במערכת הקבצים ולכן הוא יכול לגדול ולהתכווץ. רשומות גדולות מ-inodes ויכולות להכיל גם את התחלת הקובץ עצמו (ולא רק מידע עליו). גישה זו מאפשרת לקבצים קטנים להישמר בתוך רשומה וע"י כך להקטין את זמן הגישה.

בדוס וחלונות 95/98 יש מבנה פשוט יותר הנקרא **FAT** ומכיל טבלה אחת של מצביעים הממפה את כל הקבצים במערכת. הטבלה שמורה בדר"כ בתחילת הדיסק או המחיצה והיא מכילה כניסה עבור כל בלוק במערכת הקבצים. קובץ מיוצג ע"י מספר הבלוק הראשון שלו. **הבלוקים של קובץ שמורים כרשימה מקושרת בטבלת המצביעים**. כל הבלוקים שאינם מוקצים מיוצגים ברשימה נוספת. החסרונות של FAT הם זמן ארוך לגישה ישירה לקבצים גדולים (כי יש צורך לסרוק את רשימת הבלוקים של הקובץ עד מציאת הבלוק הרצוי) וקושי לממש מדיניות אפקטיבית של הקצאת בלוקים לקבצים.

מדיניות הקצאת בלוקים לקבצים

מדיניות זו משפיעה על זמני ההמתנה להזת הראש הקורא/כותב וסיבוב הדיסק ולכן משפיעה באופן דרמטי על ביצועי מערכת הקבצים. מדיניות מוצלחת תנסה לשאוף להקצות קצפים פיזיים ארוכים של בלוקים עבור רצפים

ארוכים בקובץ (כיוון שלרוב תוכניות קוראות קבצים שלמים). גם אם לא ניתן להקצות ברצף, צריך לשאוף להקצות בלוקים הקרובים פיזית. בנוסף, עליה לשאוף להקצות קבצים שמשמשים בהם יחד (בפרט מאותו מדריך או שנכתבו בסמיכות זמנים) במיקום קרוב על הדיסק.

בעבר מערכות קבצים שמרו את זהות הבלוקים החופשיים ברשימה מקושרת (שנשמרה בבלוקים החופשיים עצמם בכך שכל בלוק חופשי הכיל מצביע לבלוק החופשי הבא). במבנה נתונים כזה, מקצים בלוקים לקבצים מתחילת הרשימה (כדי להקצות במהירות) ולכן אינו מאפשר מימוש מדיניות הקצאה מתוחכמת. לאחר זמן מה, סדר הקצאת הבלוקים הוא שרירותי (כיוון שתלוי בסדר מחיקת הקבצים). במבנה כזה לאחר זמן מה שלפית נתונים מהדיסק מהווה עד כדי 4% מזמן הגישה אליו (לעומת 100%-50% במערכות מתוחכמות). במבנה כזה, ניתן לשפר מעט את הביצועים על ידי סידור מחדש של מערכת הקבצים באמצעות **איחוי** אך זהו תהליך יקר.

מדיניות נבונות של הקצאת בלוקים:

Fast File System (FFS) – מערת זו (העיקרית של גרסת BSD ביוניקס) מחלקת את הדיסק לקבוצות של **גלילים רצופים**, בניסיון להקצות נתונים קשורים זה לזה בקבוצה אחת כדי למזער את זמן ההמתנה להזת הראש. בכל קבוצה יש עותק של superblock (מבנה נתונים קטן ששומר פרמטרים של מ"ק הספציפית כמו גודל בלוק), מאגר של inodes, מערך סיביות לסימון בלוקים פנויים ומידע אחר. גודל מאגר ה-inodes תלוי בגודל קבוצת הגלילים ע"פ מפתח של inode אחד ל-2Kb נתונים (חוסך בזבוז מקום ע"פ ניסויים שבוצעו).

מערכת FFS מבצעת מדיניות דו לבית להקצאת בלוקים לקובץ חדש או להמשך קובץ קיים: תחילה, נבחרת קבוצת גלילים בה יוקצו הבלוקים. אם הקובץ קיים, יש בגליל שבו הוא נמצא מקום והקובץ עצמו תופס פחות מ-25% מהגליל (כדי לאפשר גם לאחרים לגדול וכן כיוון שבקריאת מידע רב הזזת הראש לגליל אחר פחות משפיעה), נבחר גליל זה. אחרת, נבחר גליל אחר עם מקום רב פנוי (כדי לאפשר לקובץ לגדול בו). שנית, הבלוק שיוקצה בקבוצה הנבחרת יהיה הבלוק הקרוב ביותר (סיבובית) לבלוק האחרון בקובץ (אם הוא פנוי. אחרת, יש מדיניות משנית שלא מפורטת). בנוסף, משתדלת FFS להקצות, לקרוא ולכתוב רצפים רצופים של 64KB ע"מ לאפשר לדריבר לבקש מהבקר העברה של כמות גדולה של נתונים. רצפים אלו נקראים FFS צבירים (clusters).

תוצאות מדיניות הקצאה זו הינה קבצים קשורים (אותו מדריך) נשמרים קרוב, הן מבחינת ה-inodes שלהם והן מבחינת הנתונים (או תחילת רצף הנתונים בכל קובץ). כמו כן, הבלוקים של קוב מוקצים בדר"כ ברצפים של מס' MB ששמורים ברצף סיבובי אופטימלי. בנוסף, זו גישה מאוד יעילה לבלוקים של 64KB נתונים או יותר משום שכאלה נשמרים ברצף על הדיסק.

ע"פ נסיונות, FFS מצליחה להעביר נתונים בקצבים של 50-100% מהקצב המקסימלי של הדיסק (100% נמדד כאשר תהליכים ניגשים בעיקר לבלוקים גדולים). מדיניות הקצאה של FFS טיפוסית למ"ק מתקדמות. עם זאת, במצבים מסויימים מערכות אלו אינן מספקות ביצועים טובים כאשר תהליכים רבים כותבים בו"ז לקבצים שונים (ואז צריכה להזיז את הראש בתכיפות) או כאשר יוצרים ומוחקים קבצים בתכיפות.

מערכת הקבצים היומנאית (LFS) – גם מערכת זו היא חלק מיוניקס. הרעיון הבסיסי בהקצאת הנתונים שלה היא בהתייחסות אל הדיסק כמגילה אינסופית שנתונים נכתבים בסופה (גם שכתובים). כיוון שכל הכתיבות מבוצעות בסוף, אין צורך להזיז את הראש בעת כתיבה. בנוסף, תחת ההנחה שמה שנכתב יחד צפוי להיקרא יחד, גם קריאות יבוצעו מאזור קטן בדיסק. לפי רעיון זה, כאשר משכתבים בלוק, הוא נכתב לסוף המגילה. לכן, יש צורך לעדכן גם את ה-inode שלו (ומכאן שגם הוא ייכתב בסופה). ככלל כל הבלוקים המובילים לבלוק הנתונים, ה-inode ובלוקי המצביעים יכתבו עם הבלוק בסוף. כיוון שמיקום ה-inode משתנה בכל שכתוב, מתייחסים לכל קובץ באמצעות מספר לוגי של ה-inode. מיפוי המספרים הלוגיים נשמר בקובץ מיוחד (מהבחינה שיש דרך קבועה למצוא את ה-inode שלו. משאר הבחינות הוא קובץ רגיל). בכל פעם שכותבים בלוק בסוף המגילה, כותבים גם לאיזה קובץ הוא שייך ומה מיקומו ברצף הנתונים של הקובץ.

מ"ק אינה כותבת לדיסק בכל פעם שתוכנית מבקשת לכתוב נתונים (אלא למבנה ביזרון הפיזי). נתונים ירשמו לדיסק כאשר יש מספר גדול של בלוקים מלוכלכים, עובר זמן קבוע או כאשר יש פקודה מפורשת של תוכנית. בעת כתיבה נכתבים כל הבלוקים המלוכלכים למגילה ממוינים על פי קובץ ומיקום בו.

כדי להתייחס לדיסק כמגילה אינסופית, LFS מחלקת את הדיסק למקטעים בגודל כמיליון בתים. המגילה היא רשימה מקושרת של מקטעים כאלו. החלק הכתוב של המגילה מסתיים באמצע או סוף אחד המקטעים ברשימה (וכל הבאים אחריו ריקים לחלוטין). החלק הכתוב מכיל בלוקים חיים ובלוקי זבל (ששוכתבו ואין בהם צורך). מ"ק חייבת להבטיח שיהיו בסוף הרשימה מקטעים ריקים ע"מ שתמיד יהיה ניתן לכתוב בסופה. היא עושה זאת באמצעות Cleaner המשכתב בלוקים חיים לסוף המגילה וכמסיים לשכתב את כל הבלוקים החיים במקטע משרשרת אותו לסוף. כדי לזהות אם בלוק "חי" או "מת", ה-cleaner משתמש במידע על מיקום הבלוק בקובץ ומבקש ממערכת הקבצים למפות מיקום זה. אם המיפוי מצביע על הבלוק הנוכחי – זהו בלוק חי, אחרת מת. על מנת להשיג ביצועים טובים, ה-Cleaner צריך לנקות מקטעים שרובם זבל ולשם כך קיים מבנה נתונים ייעודי.

נפילות והתאוששות

מערכת קבצים היא מבנה נתונים מורכב. מצב בו המבנה ממלא את כל התנאים נקרא מצב קונסיסטנטי. מצבים לא קונסיסטנטיים כוללים: בלוק פיזי מסומן כפנוי אך ממופה כחלק מקובץ, בלוק הממופה כחלק מיותר מקובץ אחד, בלוק שאינו ממופה בשום קובץ ואינו מסומן כפנוי, מצביע במדריך המצביע על inode המסומן כפנוי ו-inode ללא מצביעים אך שאינו מסומן כפנוי. מ"ק עוברת בין 2 מצבים עקביים דרך מצבים שאינם עקביים (כיוון

שנדרשת יותר מפעולה אחת כדי לעבור ממצב למצב). שגרות המטפלות במ"ק פועלות נכון רק אם מבנה הנתונים עקיב. במצב ללא תקלות במ"ה, ניתן להבטיח שכל שגרה מוצאת את מבנה הנתונים במצב עקיב על ידי שימוש במנעולים. עם זאת, אם מ"ה, המחשב או הדיסק מפסיקים לפעול בשל תקלה בזמן שמ"ק אינה עקיבה, מבנה הנתונים ידרוש תהליך **התאוששות (Recovery)** כדי לחזור למצב עקיב. כדי לדעת אם מ"ק היא עקיבה, קיימת סיבית מיוחדת הנבדקת בכל פעם שהמערכת עולה (לפני כניסה למצב לא עקיב מסמנים באמצעות הסיבית את הכניסה ולכן אם תבצע נפילה באמצע, כאשר המערכת תעלה בדיקת הסימון תאותת על חוסר עקיבות). ישנן מספר גישות להחזרת מערכת קבצים למצב עקיב:

מערכות עם כתיבה זהירה – מערכות שמבצעות את כל הכתיבות למ"ק בסדר המבטיח שכל מצבי הביניים הלא עקיבים יהיו מצבים בהם היא יכולה להמשיך לפעול (לא נוצר חוסר עקביות חמור). בנוסף, מערכות אלו מבצעות פעולות שמשנות את מבנה הנתונים בדיסק מייד (פעולות שמשנות את סדר מבנה הנתונים יבוצעו על פי סדר ותהליך שיבקש להשתמש במ"ק יאלץ להתין עד ביצוע כל הפעולות הנדרשות כדי להבטיח שמירה על הסדר). בעיות שעלולות להיווצר הן בעיקר אובדן משאבים עקב בלוקים או inodes פנויים שאינם מסומנים ככאלו.

כאשר מערכת כזו חוזרת לפעילות במצב לא עקיב היא יכולה להתחיל מייד לפעול ולאפשר גישה לקבצים. כיוון ששום מצב חמור לא יכול היה להיווצר, שגרות הטיפול בנתונים יפעלו נכון ומקביל יורץ תהליך תיקון ברקע שתפקידו להחזיר את המערכת למצב עקיב.

הצורך לבצע כתיבות מייד ולפי סדר מסוים מגביל את ביצועי מערכות אלו בעיקר כיוון שמונע ממ"ה לתזמן גישות לדיסק בצורה יעילה (למשל ע"י c-look). בעיות הביצועים חמורות בעיקר בסביבות בהם נוצרים ונמחקים קבצים רבים (כי פעולות אלו עשויות לגרום לחוסר עקיבות). אם רוב הגישות הן לנתונים, הביצועים טובים.

מערכות עם עדכונים רכים (Soft Updates) – פועלת בצורה דומה למערכת כתיבה זהירה אך ללא השהייה של מערכת הקבצים ועדכון הדיסק מייד. במקום זאת, בכל פעם שתהליך מבצע פעולה שמשנה את מבני הנתונים בדיסק, מערכת הקבצים מוסיפה את העדכונים שצריך לבצע בבלוקים בדיסק למבנה נתונים בזכרון. עבור כל עדכון כזה, מ"ה שומרת במבנה הנתונים את זהות העדכונים שחייבים לבצע לפני העדכון הזה כדי למנוע מצב בעייתי בדיסק. העדכונים נשמרים במבנה מיוחד ואפילו לא מבוצעים על העותק של הבלוק בזיכרון.

כאשר אלגוריתם תזמון מחליט לכתוב בלוג לדיסק, מערכת הקבצים סורקת את מבנה הנתונים של העדכונים שיש לבצע ומוצאת את כל העדכונים הדרושים (ועדכונים שיש לבצע לפנייהם וכך הלאה). היא מבצעת אותם בעותק של הבלוק בזיכרון וכותבת אותו לדיסק. העותק נשמר בזיכרון כך שניתן יהיה לבצע עליו עדכונים נוספים בהמשך שלא ניתן היה לבצע כעת (כי יש עדכונים שצריך לבצע לפנייהם).

מערכות אלו פותחו בזמן האחרון ואינן בשימו נרחב. עם זאת, ע"פ ניסויים נראה כי הן פועלות היטב.

מערכות עם כתיבה עצלה – מערכות אלו כותבות בלוקים מהזיכרון לדיסק בעיקר משיקולי ביצועים. שיקולי עקביות הן משניים וההתייחסות היחידה אליהן היא שכל פרק זמן קבוע כל הבלוקים המלוכלכים נכתבים לדיסק. קריסה של המערכת ברוב המקרים משאיר מ"ק לא עקבית הדורשת תהליך תיקון לפני חזרה לעבודה. מ"ק אלו מספקות ביצועים טובים אך לאחר נפילה לוקח זמן רב יותר עד שהמחשב חוזר לפעולה (יכול לקחת מספר שעות – בהתאם לגודל מערכת הקבצים).

מערכות קבצים מתאוששות – להשלים!

שימוש בזיכרון לא נדיף – מערכות מסוימות משתמשות בזיכרון לא נדיף (NVRAM) כדי להטמין בלוקים של נתונים. זיכרונות כאלו צריכים לשרוד נפילות עקב תקלות תוכנה ובנוסף צריכים להיות מהירים באופן משמעותי מדיסקים. הזיכרונות יכולים להיות מבוססים על זיכרונות נדיפים מגובים בסוללה או אל פסק או על זיכרונות מגנטיים מהירים. בזיכרון כזה ניתן לשמור בלוקים של מדריכים ושל מצביעים כדי לשמור על עקיבות מבנה הנתונים. אם הזיכרון מספיק גדול, ניתן לשמור בו את הבלוקים של הנתונים. העמידות שהזיכרון מספק במקרה כזה לכתיבות מאפשר לוותר על כתיבה תקופתית של בלוקים מלוכלכים לדיסק (מה שמשפר מאוד את הביצועים). ניתן גם לשמור במערכות כאלו את היומן של מערכות מתאוששות, כדי להבטיח עמידות של יצירת ומחיקת קבצים ללא ביצוע כתיבות ליומן בדיסק לפני שקראות מערכת חוזרות.

עדכון של המשך הסיכום (אם יהיה) יפורסם באתר © u.multinet.co.il

ביבליוגרפיה

תוכן שיעורי הקורס "מערכות הפעלה" פרופ' חזי ישרון, אביב 2009.
"מערכות הפעלה" \ סיון טולדו בהוצאת אקדמון.