1.4.4 מודל שרת-לקוח עמוד 23

vm\370 השיג הרבה פשטות ע״י העברת חלק גדול של קוד מע׳ ההפעלה המסורתית (הוציא את המחשב המורחב) אל רובד גבוה יותר, cms . למרות זאת vm\370 עצמו עדין תוכנית מורכבת כי סימולציה של כמה 370 מדומים היא לא קלה (במיוחד אם אתה רוצה יעילות מתקבלת על הדעת).

מגמה במערכות הפעלה מודרניות היא לקחת את הרעיון של הזזת קוד למעלה אל רובד גבוה אפילו רחוק יותר, ולהזיז כמה שאפשר רחוק יותר ממערכת ההפעלה, ולהשאיר גרעין מינימלי. הגישה המקובלת היא לבצע את רוב פעולות מערכת ההפעלה בתהליכי משתמש. כדי לבקש שרות, כמו קריאה של בלוק מקובץ, תהליך משתמש (כרגע ידוע כתהליך לקוח) שולח את הבקשה אל שרת תהליכים, אשר מבצע את הפעולה ושולח חזרה תשובה. בדגם הנראה באיור 1-12 כל מה שהגרעין מבצע הוא תקשורת בין לקוח ושרתים. ע״י הפרדת מערכת ההפעלה למעלה, לחלקים כל אחד מטפל בצד אחד של המערכת, כל חלק נעשה קטן וקל לטיפול. יתר על כן, היות וכל השרתים פועלים כתהליך משתמש ולא בתהליך גרעין, אין להם גישה ישירה אל החומרה. כתוצאה, אם באג בשרת קבצים מופעל, שרות הקבצים יכול להתמוטט, אבל לא יביא, בדרך כלל לקריסת כל המחשב. יתרון נוסף למודל שרת לקוח הוא ההתאמה לשימוש במערכת מבוזרת (ראה איור 1-13). אם לקוח מתקשר עם שרת על ידי שליחת הודעה, הלקוח לא צריך לדעת האם ההודעה מטופלת באופן מקומי במחשב שלו, או שמא היא נשלחה ברשת אל שרת רחוק יותר. בכל שנוגע ללקוח, אותו דבר קורה בשני המקרים: בקשה נשלחה ותשובה נשלחה חזרה. התמונה צוירה על הגרעין המטפל רק בהעברת הודעות מלקוחות אל שרתים וחזרה ולא ראלי לחלוטין.

עמוד 24

כמה מפעולות מערכות הפעלה (כמו הטענת פקודות אל התקני קלט\פלט הפיזיים) הם קשים, אם לא בלתי אפשרי לביצוע מתוכנית המשתמש. הי שני דרכים לטפול בבעיה זו. אחת מהן היא שיהיו כמה תהליכים שרת קריטים אשר ממש רצים במתחם הגרעין, עם גישה מלאה לחומרה, אבל עדיין לתקשר עם תהליכים אחרים ע״י שימוש במכניזם רגיל של הודעות.

הדרך השניה היא לבנות מכניזם מינימלי בתוך הגרעין, אבל להשאיר את מדיניות ההחלטה לשרתים במרחב המשתמש. לדוגמא, אם הגרעין יזהה שהודעה נשלחה לכתובת מסוימת ומתכוונת לקחת את תוכן ההודעה ולהדפיס אותה באוגרי התקני הקלט\פלט עבור דיסק כלשהו, לאתחל קריאת דיסק. בדוגמא זו הגרעין אפילו לא בודק את מספר הביטים לבדוק אם ישנן שגיאות בתוכן. הגרעין, יעתיק בדיוק את ההודעה לדיסק של אוגרי ההתקן. (ברור שתוכנית כלשהי למניעת ביצוע הודעה כזו צריכה להתבצע). החלוקה בין המכניזם למדיניות היא גישה חשובה. והיא מופיעה יותר ויותר במערכות הפעלה בהקשרים שונים.

1.5 ראשי פרקים של שאר הספר.

מערכות הפעלה עברו דרך ארוכה מאז 1950. בבסיס תוכננו לשלוט במחשב יחיד, וכיום לעיתים קרובות משתמשים בהם לאוספים רבים של מחשבים הקשורים יחדיו ברשת. מערכות ההפעלה החדשות נקראות רשת או מערכת הפעלה מבוזרת כפי שהוזכר קודם, אולם לישנות אין שם כולל.

סוגים שונים בשימוש, כולל מעבד מרכזי יחיד, cpu יחיד, או אפילו מערכות הפעלה מסורתיות. בכל המקרים הכוונה למערכת הפעלה המנהלת מחשב יחיד בעצמה. כהערה צדדית יש להזכיר שיש גם מערכת הפעלה עבור רב מעבד אשר שולט במערכות משולבות המכילות שניים או יותר cpus. היות והם שלב ביניים ממערכת של מעבד בודד אל מערכת מבוזרת ויש בהם אספקטים של שניהם נלמד אותם בהמשך הספר. שניהם, מעבד בודד, ומערכת מבוזרת הם חשובים. הראשון עדיין נפוץ יותר אולם הדבר משתנה במהירות. עד לשנת 2000 צפוי שחלק נכבד מהמחשבים שבתעשיה, אקדמיה

והממשלה יחוברו ברשת. מסיבה זו אנו מאמינים שמערכות הפעלה מבוזרות הם הדרך לעתיד והשקפה זו משתקפת בספר.
החלק הראשון של הספר הוא על מערכת הפעלה עבור מעבד יחיד. החלק השני הוא על מערכות מבוזרות. אפשר לקרוא את החלק הראשון מבלי השני, אולם הפרקים על מערכות מבוזרות מניחות מראש שהקורא מכיר את נושא מערכות הפעלה עבור מעבד יחיד.
מערכות הפעלה מחולקות בדרך כלל ל- 4 חלקים עיקריים: תהליכי ניהול, ניהול זיכרון, ניהול קבצים, וניהול התקני קלט\פלט. ארבעת הפרקים הבאים עוסקים בארבעת נושאים אלו. הפרקים לאחר מכן עוסקים: פרק 7 עוסק ביוניקס, ופרק 8 עוסק בדוס.

ת ה ל י כ י ם

אנחנו ניכנס כעת ללימוד מפורט איך מערכת הפעלה מעוצבת ובנויה. הרעיון המרכזי בכל מערכת הפעלה הוא התהליכים: שהם הפשטה של תוכנית עובדת. כל השאר תלוי ברעיון זה, וחשוב מאוד שמתכננן מערכת הפעלה (וסטודנט) ידע מהו תהליך, מוקדם ככל האפשר. בפרק זה אנו נשתמש לפעמים ביוניקס כדוגמא היות ויש לו שיטה פשוטה אולם בעלת כוח, לניהול תהליכים.

2.1 הכרת תהליכים

כל המחשבים המודרניים יכולים לבצע מספר פעולות בו זמנית. בעת ביצוע תוכנית משתמש, מחשב יכול לקרוא גם כן מהדיסק ולהדפיס במסוף או במדפסת. במערכות רב-תכנותיות, ה-cpu גם עובר ותוכנית לתוכנית כשהוא מריץ על אחת עשרות או מאות של אלפיות שניה. כאשר, למען הדיון בזמן כלשהו ה-cpu מריץ תוכנית אחת בלבד, בזמן של שניה אחת הוא יכול לעבוד על מספר תוכניות, ולכן ניתן למשתמש אשליה של מקביליות. לעיתים אנשים מדברים על מקביליות כשהכוונה היא למעברים מהירים קדימה ואחורה של ה-cpu בין תוכניות, להשוות זאת עם מקביליות אמיתית בחמרה של ה- cpu כאשר התקן אחד או יותר של קלט\פלט פועל. שמירת מסלולים של הרבה פעולות מקביליות הוא קשה לביצוע. לכן מתכנני מערכת הפעלה במשך השנים פתחו מודל ההופך את המקביליות לקלה יותר לביצוע. מודל זה הוא נושא הפרק.

2.1.1 דגם התהליכים.

בדגם זה, כל התוכנות שרצות במחשב לעיתים קרובות ומכילות את מערכת ההפעלה, מאורגנים לכמה תהליכים רצופים, או רק תהליכים בקיצור. תהלים הוא רק תוכנית פועלת, כולל הערך הנוכחי של מונה התוכנית, האוגרים, והמשתנים. באופן עקרוני, לכל תהליך יש cpu מדומה שלו. במציאות, כמובן, המערכת, קל יותר לחשוב על אוסף של תהליכים הפועלים במקביליות, ואז לנסות לשמור מסלול של איך ה- cpu עובר מתוכנית לתוכנית. מעברים תכופים אלו קדימה ואחורה נקראים רב-תכנות (multiprogramming), כפי שראינו בפרק קודם. באיור 1-2 נראה מחשב המפעיל 4 תוכניות בזכרון.

באיור 1-2 אנו רואים כיצד אנו מפשטים ל- 4 תוכניות, כאשר לכל אחת תרשים זרימה משלו לבקרה (מונה התוכנית האישי שלו) וכל אחד פועל ללא תלות באחרים. באיור 1-2 אנו רואים שבצפיה ארוכה מספיק, כל התהליכים יצרו תהליכים אולם בכל רגע, רק תהליך אחד באמת פועל.

עם ה- cpu עובר קדימה ואחורה בין תהליכים, שיעור הביצוע בין תהליכים הוא לא אחיד וכנראה לא יעיל אם אותו תהליך יפעל שוב. לפיכך, אין לתכנת תהליכים בהנחה קבועה של זמנים. לדוגמא, ניקח תהליך קלט\פלט שמאתחל סרט מגנטי לפעולה, לביצוע לולאה 1000 פעמים כדי שהסרט יכנס למהירות הנכונה, ואז לאשר את הפקודה לקרוא את הרשומה הראשונה. אם ה-cpu יחליט לעבור לתהליך אחר בעת ביצוע הלולאה, תהליך הסרט אולי לא ירוץ יותר עד אחרי שהרשומה הראשונה תעבור את הראש הקורא. כאשר לתהליך יש קטע קריטי שכזה ארועים מסוימים מוכרחים להתבצע בזמן מסוים של אלפיות שניה, יש לבצע הערכות מיוחדות על מנת להבטיח את הביצוע. אולם, באופן רגיל רוב התהליכים לא מושפעים מביצוע מספר תוכניות ע"י ה- cpu או הזמן הלא מוחלט של תהליכים שונים.

ההבדלים בין תהליך ותוכנית הוא עדין אולם קריטי. אנלוגיה אולי תבהיר את הנקודה.

נניח שמדען מחשבים אופה לביתו עוגת יום-הולדת. יש לו מרשם לעוגה ומטבח מאובזר היטב עם הקלט הדרוש: קמח, ביצים, סוכר, מקל וניל וכו'. באנלוגיה זו המרשם הוא התוכנית, מדען המחשב הוא המעבד, ומרכיבי העוגה
הם הנתונים. התהליך הוא הפעילות של האופה בקריאת המרשם, ואפית העוגה.
כעת תדמינו שבנו של איש המחשבים נכנס בבכי, אומר שהוא נעקץ על ידי דבורה. איש המחשבים זוכר היכן היה במרשם (המצב הנוכחי של התהליך נשמר), יוצא עם ספר עזרה ראשונה ומתחיל לבצע את ההוראות הרשומות. כאן אנו רואים מעבד שעובר מתהליך אחד לתהליך עם עדיפות גבוהה יותר, לכל אחד תוכנית אחרת (מרשם וספר עזרה ראשונה). כאשר העקיצה טופלה איש המחשבים חוזר לעוגה, וממשיך מהנקודה שבה הפסיק.
רעיון המפתח כאן הוא שתהליך הוא פעילות מסוג כלשהו, זוהי תוכנית, קלט, פלט ומצב. מעבד יחיד יכול להתחלק בין כמה תהליכים, עם כמה אלגוריתמים לחישוב מתי להפסיק פעולת תהליך אחד ולשרת תהליך אחר.

<u>היררכית תהליכים</u>

מערכת הפעלה שתומכת ברעיון חייבת לספק כמה דרכים ליצור מה שהתהליכים צריכים. במערכת פשוטה מאוד, או במערכות המתוכננות לבצע בקשה אחת ניתן שכל התהליכים שידרשו מתי שהוא יהיו נוכחים בעת שהמערכת עולה. ברוב המערכות, בכל מקרה דרושה דרך ליצור ספרית תהליכים הדרושים במהלך פעולה. ביוניקס, תהליכים נוצרים על ידי קריאת מערכת fork ת כאשר היא יוצרת העתק מדויק של התהליך הנקרא. אחרי קריאת המערכת fork ההורה ממשיך לרוץ במקביל לילד. ההורה יכול להסתעף לעוד ילדים כך שבכל רגע קיים אפשר שיהיו מספר ילדים פועלים. הילדים יכולים גם כן לבצע סיעוף, כך שאפשר לקבל עץ תהליכים בעומק שרירותי.
ב-ms-dos קריאות מערכת קיימות כדי לטעון קובץ בינארי לזיכרון ולבצע אותו כתהליך בן. בניגוד ליוניקס בדוס תהליך ילד משעה את ההורה עד גמר ביצוע תהליך הילד, כך שהורה וילד לא פועלים במקביל.

<u>מצב תהליך</u>

למרות שכל תהליך הינו ישות בלתי תלויה, עם מונה תוכנית שלו ומצב פנימית, תהליכים לעיתים קרובות צריכים תקשורת עם תהליכים אחרים. תהליך אחד יכול ליצור פלט כלשהו שיהווה קלט לתהליך אחר. בפקודות מעטפת
cat chapter1 chapter2 chapter3 | grep tree התהליך הראשון מפעיל את cat משרשר שלושה קבצים. התהליך השני מפעיל את grep, בוחר את כל השורות המכילות את המילה tree.

תלוי במהירות היחסית של שני התהליכים (ושניהם תלויים במורכבות היחסית של התוכניות וכמה זמן יש למעבד לכל אחד מהם), יתכן ויקרה שתהליך grep מוכן לפעולה אולם אין לו קלט לביצוע. הוא חייב להחסם עד אשר יש קלט זמין.
כאשר תהליך נחסם, זה כי באופן הגיוני הוא אינו יכול להמשיך, בדרך כלל כי הוא ממתין לקלט שאינו זמין. כמו כן אפשר שיקרא שתהליך מוכן להמשיך באופן עקרוני אולם נעצר כי מערכת ההפעלה החליטה שהמעבד יעבוד עם תהליך אחר בינתיים. שני תנאים אלו שונים בתכלית השינוי. במקרה הראשון ההשעיה היא תוצאה של בעיה (אתה לא יכול לבצע פקודת משתמש לפני שהיא הוקלדה). במקרה השני, זוהי טכניקה של המערכת (אין מספיק מעבדים כדי לתת לכל תהליך מעבד פרטי משלו). באיור 2-2 רואים מצב של תרשים המראה שלושה מצבים שהתהליך יכול להיות בהם:

1. פועל (באופן ממשי פועל במעבד ברגע זה).
2. מוכן (מוכן לפעולה. מופסק באופן זמני כדי לאפשר לתהליך אחר לרוץ).
3. מופסק. (לא מסוגל לרוץ עד אשר אירוע חיצוני יקרה).

באופן עקרוני שני המצבים הראשונים דומים. בשני המקרים התהליך מוכן לפעולה, אלא שבשני באופן זמני המעבד לא פנוי לכך. המצב השלישי שונה משני הראשונים כי כאן התהליך לא יכול לפעול אפילו אם המעבד פנוי.
ארבעה מעברים אפשריים בין שלושת המצבים. מעבר 1 קורה כאשר תהליך מגלה שאינו יכול להמשיך. בכמה מערכת התהליך חייב לבצע קריאת מערכת, block כדי להגיע למצב חסימה. שכיח יותר, כאשר תהליך קורא מצינור או קובץ מיוחד (מסוף למשל) ואין קלט זמין, התהליך אוטומטית נחסם.
מעבר 2 ו - 3 קורים ע״י בקרת זמנים של התהליך (חלק ממערכת ההפעלה), מבלי שהתהליך אפילו ידע עליהם. מעבר 2 קורא כאשר בקרת זמנים מחליטה שתהליך מסוים בוצע זמן ארוך מספיק והגיע הזמן לתת לתהליך אחר זמן מעבד. מעבר 3 קורה כאשר כל התהליכים האחרים קבלו זמן מעבד והגיע זמנו של התהליך הראשון, שוב. נושא לוח הזמנים המחליט איזה תהליך יבוצע ולכמה זמן הוא חשוב. אנו נחזור אליו במהלך הפרק.

עמוד 31

אלגוריתמים רבים הומצאו לנסות לאזן בהתמודדות עם דרישות היעילות של כל המערכת וההגינות להליך הבודד.
מעבר 4 קורה כאשר קורה תהליך חיצוני אשר תהליך המתין לו (כמו הגעה של קלט). אם אף תהליך לא פועל כרגע מעבר 3 יפעל מיד, והתהליך יתחיל להתבצע מיידית, אחרת הוא ימתין במצב היכון לזמן קצר עד אשר המעבד יהיה פנוי.
כאשר משתמשים במודל תהליך, קל יותר לחשוב מה קורה בתוך המערכת. כמה מהתהליכים מבצעים תוכניות הנושאות פקודות שהוקלדו על ידי המשתמש. תהליכים אחרים הם חלק מהמערכת ומטפלים במשימות כגון בקשות לשרות עבור קבצים, או ניהול הפרטים של הרצת הדיסק או הכוננים. כאשר פסיקת דיסק קוראת המערכת מחליטה להפסיק את התהליך שמבוצע כרגע ולבצע את תהליך הדיסק, אשר היה חסום וחיכה לפסיקה. לפיכך, במקום לחשוב על פסיקות, אנו יכולים לחשוב על תהליכי משתמש, תהליכי דיסק, תהליכי מסוף וכן הלאה אשר חסומים כאשר הם מחכים למשהו שיקרה. כאשר קוראים בלוק מהדיסק או תו מוקלד, התהליך המחכה לכך עובר להיכון והוא כשיר שוב לביצוע.
זוית ראיה זו היא המקור לאיור 3-2. כאן הדרגה התחתונה של מערכת ההפעלה הוא בקר הזמנים, עם תהליכים שונים מעליו. כל הטיפול בפסיקות, והפרטים איך בפועל להתחיל ולעצור תהליך מוסתרים בבקר הזמנים, שהוא קטן יחסית. שאר מערכת ההפעלה בנויה בצורת תהליך.

2.1.2 ביצוע תהליכים

כדי לבצע מודל של תהליך מערכת ההפעלה שומרת טבלה (מערך של מבנים), הנקראת טבלת תהליכים, עם כניסה אחת לכל תהליך. הכניסה שומרת מידע על מצב התהליך, על מונה התוכנית, מצביע מחסנית, מיקום הזיכרון, ומצב הקבצים הפתוחים, הוא רושם את המידע וכל דבר אחר על התהליך שיש לשמרו כאשר מצב תהליך משתנה ממבוצע למוכן לביצוע, כדי שהתהליך יופעל שוב מאוחר יותר כאילו מעולם לא הופסק.
אף על פי שהשדות המדויקים בטבלת התהליכים משתנה ממערכת למערכת, בדרך כלל חלק יעסקו עם ניהול תהליכים, חלק עם ניהול זיכרון ואחרים עם מערכת קבצים. איור 2-4 מראה כמה מהשדות הנפוצים ביותר כרגע במערכות יוניקס.

עמוד 32

כעת אחרי שהתבוננו בטבלת התהליכים, אפשר להסביר קצת יותר כיצד האשליה של הרבה תהליכים סידרתיים נשמרת במחשב עם מעבד אחד והרבה התקני קלט\פלט. כל אחד מהתקני קלט\פלט (דיסק לא קשיח, דיסק קשיח, שעונים, מסופים) ממוקם בתחתית הזיכרון ונקרא מערך פסיקות. הוא מכיל את הכתובות של שגרות השרות. נניח שתהליך משתמש מספר 3 פועלת כאשר קורת פסיקת דיסק. מונה התוכנית, מילת מצב התוכנית ויתכן שאחד או יותר מהאוגרים נדחף אל מחסנית על ידי החומרה שמטפלת בפסיקות. המחשב אזי קופץ לכתובת שמצויינת במערך הפסיקות. זה כל מה שהחומרה עושה. מכאן הכל תלוי בתוכנה. שגרות השרות לפסיקות מתחילות על ידי שמירת כל האוגרים בכניסה

לטבלת תהליכים עבור התהליך הנוכחי. מספר התהליך הנוכחי ומצביע לכניסה שלו נשמרים במשתנים גלובלים כך שניתן לאתרם בקלות. אזי המידע שהופקד על ידי הפסיקה מועבר מהמחסנית, ומצביע המחסנית מכוון אל מחסנית זמנית לשימוש אחראי התהליכים. לא ניתן אפילו להביע בשפת c פעולות כגון שמירת האוגרים והכוונת מצביע המחסנית (או בכל שפת עילית אחרת) לכן הן מבוצעות על ידי שגרה קטנה בשפת אסמלי.
כאשר השגרה מסיימת את פעולתה, היא קוראת לשגרת C לבצע את העבודה האמיתית של ביצוע הפסיקה.
השלב הבא הוא להחליט איזה תהליך התחיל את בקשת הדיסק. בדרך כלל התהליך יופסק אחרי איתחול הבקשה, כך שעתה עליו להתעורר שוב. המצב של התהליך כרגע משתנה מחסום למוכן ובקרת הזמנים נקראת.
אם התהליך פועל שפועל אחר כך תלוי באלגורתם לבקרת זמנים. אנו יודעים לבטח כי לפחות שני תהליכים כרגע מוכנים: התהליך שהתחיל הבקשה לדיסק והשני שהופסק. בקר הזמנים הוא שקובע מי חשוב יותר. ביוניקס לדוגמא, תהליכים הקשורים בפלט\קלט הם בעלי קדימות גבוהה מאשר תהליכים הקשורים במעבד, כך שניתן לאשר את הבקשה הבאה בנושא קלט\פלט במהירות. גישה זו תומכת במקביליות, ומנצלת את המעבד והדיסק יחדיו. אולם למערכות אחרות גישה שונה, למשל, הדגשת ההגינות על היעילות.

כדי לחזור לתהליך שנבחר מחדש, שגרת C נקראת על ידי קוד לפסיקה בשפת אסמבלי שוב, וקוד בשפת אסמבלי טוען את האוגרים ומפת הזיכרון עבור התהליך הנוכחי שמתחיל לפעול. אחראי הפסיקות ובקר הזמנים מתומצתים באייור 5-2.

1. החומרה מאחסנת את מונה התוכנית.
2. החומרה טוענת מונה תוכנית חדשה ממערך הפסיקות.
3. שגרה בשפת אסמבלי שומרת את האוגרים.
4. שגרת אסמלי מאתחלת מחסנית חדשה.
5. שגרת C מסמנת תהליך שרות כמוכן.
6. בקר הזמנים מחליט מי התהליך הבא שיפעל.
7. שגרת C חוזרת לקוד באסמבלי.
8. שגרה בשפת אסמבלי מתחילה להפעיל את התהליך הנוכחי.

2.2 תקשורת בין תהליכים

לעיתים קרובות תהליך זקוק להתקשר עם תהליך אחר. לדוגמא, בצינור מידע של המעטפת הפלט של תהליך ראשון חייב להיות מועבר לתהליך שני, וכן הלאה. לפיכך יש צורך בתקשורת בין תהליכים, עדיף בצורה בנויה היטב ללא שימוש בפסיקות. בפרק הבא נדון בכמה מהנושאים הקשורים בתקשורת הפנימית או בקיצור IPC.

2.2.1

בכמה מערכות הפעלה, תהליכים שעובדים יחדיו לעיתים קרובות חולקים מחסן משותף שכל אחד יכול לקרוא ולכתוב. המחסן המשותף יכול להיות בזיכרון המרכזי או אפשר שיהיה קובץ משותף. המיקום של הזיכרון המשותף לא משנה את אופי התקשורת או הבעיות שצצות. כדי לראות כיצד תקשורת בין תהליכים פועלת בא נתבונן בדוגמא פשוטה אולם נפוצה, סליל מודפס. כאשר תהליך רוצה להדפיס קובץ, הוא מכניס את שם הקובץ לספרית סליל (SPOOLER). תהליך אחר, PRITER DAEMON (שד המדפסת) בודק תקופתית אם ישנם קבצים להדפסה, אם כן הוא מדפיס אותם ומעביר את שמם מספריה זו. תארו לעצמכם שלספרית הסליל שלנו יש מספר רב של חריצים, ממוספרים החל מ- 0 כאשר כל אחד יכול לשמור שם קובץ. כמו כן תארו לעצמכם שישנם שני משתנים משותפים, OUT, אשר מצביע על הקובץ הבא בתור להדפסה, ו- IN אשר מצביע על החריץ הבא הפנוי בספריה. שני משתנים אלו ישמרו בקובץ בן שני מילים הזמין לכל התהליכים. ברגע מסוים חריץ 0 עד 3 ריקים (הקבצים הודפסו כבר), וחריץ 4 עד 6 מלאים

(עם שמות קבצים בתור להדפסה). במקביל תהליכים A ו- B מחליטים שהם רוצים להצטרף לתור הקבצים הממתינים להדפסה. מצב זה נראה באיור 2-6. בתחום השיפוט במקום שחוקי מרפי מתאימים הדבר הבא עלול לקרות. תהליך A קורא IN ומאוחסן בערך 7 במשתנה המקומי הנקרא "החריץ הפנוי הבא". בדיוק פסיקת זמן מתרחשת והמעבד מחליט שתהליך A פעל דיו, והוא עובר לביצוע תהליך B. תהליך B קורא גם כן IN, וגם מקבל את חריץ 7, כך שהוא מאחסן את שם הקובץ בחריץ 7 ומעדכן ב-IN את חריץ 8. ואז הוא מבצע דברים אחרים.

עמוד 34

לבסוף תהליך A פועל שוב, ומאותחל מהמקום שהופסק. הוא מסתכל –ב- "חריץ הפנוי הבא", מוצא שם 7 וכותב את שם הקובץ שלו בחריץ 7, ומוחק את השם שתהליך B בדיוק רשם שם. אזי הוא מוסיף לערך "החריץ הפנוי הבא" 1 שמקבל את הערך 8 ומציין IN בחריץ 8. ספרית הסליל כרגע יציבה, כך ששד המדפסת לא יבחין בשגיאה, אולם תהליך B לעולם לא יקבל פלט. מצב כזה כאשר שני או יותר תהליכים קוראים או כותבים נתונים משותפים, והתוצאה הסופית תלויה במי פועל בדיוק נקראת "מצב תחרות". איתור שגיאות בתוכניות המכילות מצבי תחרות היא לא צחוק בכלל. תוצאתם של רוב הבדיקות היא תקינה, אולם לעיתים נדירות דברים מוזרים ולא מוסברים קורים.

2.2.2 קטעים קריטיים

כיצד להמנע ממצב תחרות? המפתח למניעת צרות כאן ובהרבה מצבים אחרים הקשורים בזיכרון משותף, קבצים משותפים, וכל דבר משותף, הוא למצוא דרך למנוע מיותר מתהליך אחד לקרוא או לכתוב בנתונים המשותפים בו זמנית. במילים אחרות, מניעה הדדית – דרך להבטיח שרק תהליך אחד ישתמש במשתנה משותף, או קובץ, לתהליך האחר לא יאופשר לבצע את אותו דבר. הקושי שלעיל נגרם כי תהליך B התחיל להשתמש באחד המשתנים המשותפים לפני שתהליך A סיים איתו. הברירה בפעולה פרימיטיבית נאותה, כדי להשיג מניעה הדדית הוא נושא תכנוני חשוב בכל מערכת הפעלה, ונושא שאנו נבחן לפרטי פרטים בפרקים הבאים.

בעית ההמנעות ממצב תחרות ניתן לנסח בדרך מופשטת. חלק מהזמן תהליך עסוק בחישוב פנימי ודברים אחרים שלא מובילים למצב תחרות. אולם לעיתים תהליך יכול להכנס לזיכרון משותף או קבצים, או לבצע קטעים קריטים אחרים שיובילו לתחרות. חלק זה של התוכנית כאשר זיכרון משותף הוא נגיש נקרא קטע קריטי. אם נוכל למנוע משני תהליכים להיות בו זמנית בקטע הקריטי שלהם נמנע ממצב תחרות.

עמוד 35

למרות שדרישה זו תמנע מצב תחרות, לא מספיק שתהליכים מקבילים ישתפו פעולה וינהלו ביעילות נתונים משותפים. אנו זקוקים ל4 תנאים עבור פתרון טוב:

1. שני תהליכים לא יהיה יחדיו בתוך קטע קריטי.
2. לא ישערו השערות לגבי מהירות או מספר המעבדים.
3. תהליך הפועל מחוץ לקטע קריטי לא יחסום תהליך אחר.
4. תהליך לא ימתין לנצח להכנס לקטע קריטי.

2.2.3 מניעה הדדית עם המתנה פעילה

בפרק זה נבחן הצעות שונות להשגת מניעה הדדית, כך שכאשר תהליך מעדכן זיכרון משותף באזור הקריטי שלו שום תהליך אחר לא יכנס לאזור הקריטי ויגרום צרות.

התרת פסיקות

הדרך הפשוטה ביותר למנוע מתהליך את כל הפסיקות עם הכניסה לאיזור קריטי, ולאפשר אותן שוב לפני היציאה. עם מניעת הפסיקות שום פסיקה לא תקרה, אחרי הכל כשהפסיקות מכובות המעבד לא יעבור לתהליך אחר. לפיכך כאשר לתהליך אין פסיקות, הוא יכול לבדוק ולעדכן את הזיכרון המשותף ללא חשש מהתערבות תהליך אחר.

גישה זו בדרך כלל לא מקובלת כי לא חכם לתת לתהליך משתמש את הכוח לכבות את הפסיקות. נניח שאחד מהם עשה זאת, ושכח להדליקם בחזרה? זה יכול להיות סוף המערכת. יותר מזה, אם למחשב יש יותר ממעבד אחד, מניעת פסיקות משפיע על מעבד אחד בלבד שמבצע את הוראות מניעת הפסיקות. כל השאר ימשיכו ויוכלו להכנס לזכרון המשותף.

מאידך, לעיתים קרובות נוח לגרעין למנוע פסיקות לכמה הוראות כאשר הוא מעדכן משתנים או רשימות. אם קורת פסיקה כאשר רשימה של תהליכים מוכנים, לדוגמא, היו במצב לא יציב, תהליך תחרות יופיע. המסקנה היא: מניעת פסיקות לעיתים היא שיטה יעילה בתוך הגרעין, אולם לא מתאימה באופן כללי למכניזם של מניעה הדדית עבור תהליכי משתמש.

משתנים נעולים

כנסיון שני, נתבונן בפתרון תכנותי. נניח שיש לנו משתנה משותף, מאותחל 0. כאשר תהליך רוצה להכנס לקטע קריטי הוא בודק תחילה את הנעילה. אם המנעול הוא 0 התהליך משנה אותו ל- 1 ונכנס לקטע קריטי. אם ערך המנעול הוא 1 התהליך ימתין עד שהוא ישתנה ל- 0.

לפיכך, 0 מציין שאין תהליך בקטע קריטי, ו- 1 מציין שיש תהליך בקטע קריטי. לרוע המזל, תהליך זה מכיל בדיוק את אותה בעיה שראינו בספרית הסליל. נניח שתהליך אחד קורא את המנעול ורואה שערכו הוא 0. לפני שהוא משנה אותו ל- 1 תהליך אחר נקרא לפעולה ומשנה את המנעול ל- 1. כאשר שניהם בקטע קריטי ביחד.

כעת, אתה עלול לחשוב שניתן לפתור את הבעיה על ידי קריאת ערך המנעול, ולפני אחסון הנתונים נבדוק אות שוב, אולם זהו לא פתרון. התחרות תקרה עתה אם תהליך ישנה את המנעול מיד אחרי שהתהליך הראשון סיים את הבדיקה השניה.

שינוי ברור

גישה שלישית לבעית המניעה ההדדית נראית באיור 2-7. קטע תוכנית זה כתוב ב- C היות ומערכות הפעלה לעיתים קרובות כתובות ב- C.

```
WHILE TRUE
  { WHILE (TRUE!=0)
CRITICAL_SECTION();
TURN=1;
NONCRITICAL_SECTION();
 }
WHILE TRUE
  { WHILE (TRUR!=1)
CRITICAL_SECTION();
TURN=0;
NONCRITICAL_SECTION();
 }
```

באיור 2-7 המשתנה TURN מאותחל ל- 0 שומר על תור מי להכנס לקטע קריטי ולבדוק או לעדכן את הזכרון המשותף. בתחילה תהליך 0 בודק את TURN מוצא 0 ונכנס לקטע קריטי. תהליך 1 גם מוצא אותו כ- 0 ולכן נכנס ללולאה שבודקת ללא הפסק האם הוא השתנה ל-1. בדיקה רצופה של משתנה בהמתנה לשינוי ערכו נקראת המתנה פעילה. ויש בדרך כלל להמנע ממנה היות והיא מבזבזת את זמן המעבד. רק כאשר צפויה המתנה קצרה ביותר מותרת המתנה פעילה.

כאשר תהליך 0 עוזב את הקטע הקריטי, הוא משנה את TURN ל- 1 כדי לאפשר לתהליך מספר 1 להכנס לקטע הקריטי. נניח שתהליך 1 סיים מהר את הקטע הקריטי כך ששני התהליכים לא נמצאים בחלק הקריטי, כאשר TURN ערכו 0. כעת תהליך 0 מבצע את כל הלולאה במהירות וחוזר לאיזור הקריטי עם TURN שערכו 1. בנקודה זו, תהליך 0 סיים את החלק שאינו קריטי וחזר לתחילת הלולאה. אולם, אין הוא יכול להכנס לקטע הקריטי עכשיו כי TURN הוא 1, אולם תהליך 1 עסוק בכלל לא בקטע קריטי. ולכן עבודה בתורות לא יעילה אם תהליך אחד מהיר יותר מהתהליך השני.

מצב זה סותר את תנאי 3 שהוצג לעיל: תהליך 0 נחסם על ידי תהליך שלא נמצא בקטע קריטי. נחזור אל ספרית ההדפסה (SPOOLER) שנדונה לעיל, אם נאחד את הקטע הקריטי עם קריאה וכתיבה בספרית ההדפסה, תהליך 0 לא יורשה להדפיס קובץ נוסף כי תהליך 1 מבצע משהו אחר. למעשה, פתרון זה דורש ששני התהליכים יתחלפו בכניסה לאיזור הקריטי. לדוגמא, בקובצי הדפסה. אף אחד מהם לא יורשה להכנס לספריה פעמיים רצופות. כאשר אלגוריתם זה מונע תחרות, הוא לא באמת מועמד רציני כפיתרון.

<u>פתרון פטרסון</u>

על ידי שילוב הרעיון של חלופה בתורות עם רעיון המשתנים הנעולים ומשתני אזהרה, מתמטיקאי הולנדי ט. דקר, היה הראשון להמציא פתרון תיכנותי עבור בעית המניעה ההדדית שלא דורש כניסה לסרוגין לקטע קריטי.

בשנת 1981 פטרסון גילה דרך פשוטה יותר להשיג מניעה הדדית, לפיכך ביצוע של פתרון דקר הוא מיושן. האלגוריתם של פטרסון נרשם באיור 2-8. אלגוריתם זה מורכב משתי שגרות שנכתבו ב-ANSI C, כשהכוונה היא שיש לספק לכל הפונקציות שמוגדרות, ובשימוש, אבות טיפוס. בדרך כלל אבות הטיפוס (PROTOTYPES) מאוגדים יחד בקובץ HEADER. (קובץ כותרת).

השורה הראשונה באיור 2-8 מכילה את כל אבות הטיפוס הללו. הפרטים הללו לא חשובים באמת כאן, לכן לא נרשמו בספר.

```
#  include “prototttypes.h”
#  define FALSE  0
#  define TRUE    1
#  define N          2

 int   turn;                                /*whose turn is it?
Int  interested[N]                          /*all values initially 0 (FALSE)*/
Void enter_region (int process)             /* process: who is entering  (0 or 1)*/
{
  int other;                                /*number of the other process*/

other = 1 - process                         /*the oppsite of process*/
interested[process] = TRUE                  /*show that you are interested*/
turn = process;                             /*set plag*/
while (turn == process && interested[other] == TRUE)     /*nukk statement*/;
}
void leave_rgion (int pricess)              /*process: who is leaving (0 or 1) */
{
  interested[process] = FALSE;              /*indicate departure from critical
region*/
}
```

<u>עמוד 38</u>

לפני שמשתמשים במשתנה המשותף (כלומר, לפני שנכנסים לקטע קריטי) כל תהליך קורא ל-enter_region עם מספר התהליך שלו 0 או 1 כפרמטר. אם צריך, הקריאה תגרום לו להמתין עד אשר בטוח להיכנס. אחרי שהתהליך סיים עם המשתנים המשותפים התהליך קורא ל-leave_region כדי להודיע שהוא סיים ולאפשר לתהליך השני להכנס אם הוא רוצה.

נתבונן כיצד הפתרון עובד. בתחילה אף תהליך לא נמצא באיזור קריטי. כעת תהליך מספר 0 קורא ל-enter_region. הוא מודיע על רצונו על ידי עדכון המערך ושינוי המשתנה turn ל- 0. מאחר ותהליך 1 לא מעונין, enter_region מוחזר מיידית. אם תהליך 1 עכשיו

יקרא ל-enter_region, הוא ימתין שם עד שתא המערך interested[0] ישתנה ל-FALSE, אירוע שיקרה רק כאשר תהליך 0 יקרא ל-leave_region.
עתה, נבדוק את המקרה ששני התהליכים קוראים יחד למשתנה enter_ergion.
שניהם יאחסנו את מספר התהליך שלהם ב-turn. מי מהם שמאחסן אחרון הוא הזוכה.
הראשון מפסיד. נניח שתהליך 1 מאחסן אחרון, אזי turi הוא 1. כאשר שני התהליכים מגיעים להצהרת while , תהליך 0 מבצע אותו 0 פעמים ונכנס לקטע הקריטי. תהליך 1 מבצע את הלולאה ולא נכנס לאיזור הקריטי.

<u>הוראות ה-TSL</u>

כעת נתבונן בהצעה שדורשת עזרה קטנה מהחומרה. למחשבים רבים, במיוחד אלו שתוכננו עם כמה מעבדים, יש הוראות TSL (TEST AND SET LOCK) שפועל בצורה הבאה: הוא קורא את תוכן מילת הזיכרון אל אוגר ואז מאחסן ערך שונה מ- 0 בכתובת הזיכרון הזו . הפעולה של קריאת מילה ואיחסון בתוכה היא פעולה בלעדית, אף תהליך אחר לא יכול להיכנס למילה עד שההוראות הסתיימו. המעבד מבצע את הוראות ה-TSL, נועל את ערוץ הזיכרון כדי לאסור על מעבד אחר מלהכנס לזיכרון עד גמר הביצוע.
כדי להשתמש בהוראות TSL אנו נשתמש במשתנה משותף, flag, כדי לתאם הוראות, ואז לקרוא או לכתוב בזיכרון המשותף. כאשר סיימנו, מצב התהליך flag מאופס תוך שימוש בהוראת MOVE רגילה.
איך ניתן להשתמש בהוראה כדי למנוע שני תהליכים מלהיכנס יחד אל הקטע הקריטי שלהם? הפתרון ניתן באיור 2-9. יש 4 הוראות תוכניות משנה בשפת אסמבלי בדויה (אבל טיפוסית). ההוראה הראשונה מעתיקה את הערך הקודם של flag לאוגר ואז משנה את flag ל- 1. אזי הערך הקודם מושווה עם 0. אם הוא לא 0, המנעול כבר נעול, כך שתוכנית רק חוזרת להתחלה ובודקת אותו שוב. מאוחר יותר הוא ישתנה ל- 0 (כאשר התהליך שנמצא כרגע בקטע הקריטי סיים את הקטע הקריטי), ותוכנית המשנה חוזרת עם מנעול מעודכן.
איפוס המנעול הוא תהליך פשוט, התוכנית רק מאחסנת 0 ב-flag. אין צורך בהוראה מיוחדת.
פתרון אחד לבעית הקטע הקריטי הוא כעת ברור. לפני כניסה לאיזור קריטי, תהליך קורא ל-enter_region, אשר מבצע המתנה פעילה עד שהמנעול משתחרר, אזי הוא משיג את המנעול וחוזר. אחרי הקטע הקריטי התהליך קורא ל-leave_region , אשר מאחסן 0 במנעול. בכל הפתרונות המבוססים על איזור קריטי, התהליך חייב לקרוא ל-enter_region ול- leave_region ברגע הנכון כדי שהשיטה תפעל. אם תהליך מרמה, המניעה ההדדית תכשל.

<u>עמוד 39</u>

<u>2.2.4</u>

שניהם, פתרון פטרסון והפתרון שמשתמש ב-TSL נכונים, אבל לשניהם יש פגם בדרישה להמתנה פעילה. ובקיצור, מה שפתרונות אלו עושים הוא זה: כאשר תהליך רוצה להיכנס לקטע קריטי, הוא בודק לראות אם הכניסה מותרת. אם לא התהליך פשוט נכנס ללולאה שממתינה עד שהכניסה תותר.
לא רק שגישה זו מבזבזת זמן מעבד, אלא שיכולה להיות לה תוצאה לא צפויה. נניח שמחשב עם שני תהליכים, H, עם קדימות גבוהה, ו-L, עם קדימות נמוכה. חוקי לוח הזמנים הם שכאשר H, במצב מוכן, הוא רץ. ברגע מסוים, כאשר L נמצא בקטע קריטי, H עובר למצב מוכן לפעולה. H מתחיל עתה בהמתנה פעילה, אבל היות וכאשר H פועל L ממתין אזי L לעולם לא יצא מהקטע הקריטי. מצב זה נקרא <u>בעיית היפוך עדיפויות</u>.
בו נתבונן עתה בתקשורת פנימית, פרימיטיבית, בין תהליכים אשר חוסמת במקום לבזבז זמן מעבד כאשר לא ניתן להיכנס לקטעים קריטיים. אחד הפשוטים הוא הזוג SLEEP ו-WAKEUP. SLEEP הוא קריאת מערכת שגורמת לקורא להיחסם, זאת עד אשר תהליך

אחר מעיר אותו. לקריאה WAKEUP יש פרמטר אחד, התהליך שיש להעיר. לחלופין, לשניהם, ל- SLEEP ול-WAKEUP יש פרמטר אחד, כתובת בזיכרון הדרוש לצורך התאמה בין SLEEP ובין /WAKEUP
כך אנו מקבלים המתנה רדומה שלא מעסיקה את המעבד.
על ידי שימוש ב-SLEEP ניתן ליצור תור של תהליכים מדומים.

בעיית היצרן- צרכן

תאור הבעיה: שני תהליכים משתמשים במאגר נתונים בעל גודל קבוע, משותף. אחד מהם היצרן, מכניס מידע אל החוץ\צץ, והשני, הצרכן, מוציא נתונים. הבעיה מתעוררת כאשר היצרן רוצה להכניס ערך לחוצץ, אבל הוא כבר מלא. הפתרון עבור היצרן הוא להרדים את עצמו, ולהתעורר כאשר הצרכן יוציא פריט אחד או יותר מהחוצץ. בדומה, אם הצרכן רוצה להוציא פריט מהחוצץ אולם מוצא את החוצץ ריק, הוא הולך לישון עד אשר היצרן יכניס משהו לחוצץ ויעורר אותו.

עמוד 40

גישה זו נשמעת פשוטה למדי, אולם היא מובילה לסוג מסוים של מצב תחרות. כדי לעקוב אחרי מספר הפריטים שבחוצץ, נצטרך משתנה בשם count. אם החוצץ יכול להכיל N פריטים, אזי הקוד של הצרכן יבדוק בתחילה לראות אם count שווה ל-N. אם כן היצרן ילך לישון. אם לא היצרן יוסיף פריט, ויגדיל את count.
הקוד של הצרכן דומה: תחילה יבדוק את count לראות אם הוא 0. אם כן הוא ילך לישון, אחרת הוא יוציא פריט, ויפחית את המונה. כל אחד מהתהליכים יבדוק גם כן אם השני צריך לישון, אם לא, הוא יעיר אותו. הקוד לשני התהליכים רשום באיור 2-10.
כדי לבטא קריאת מערכת כגון SLEEP וכגון WAKEUP בשפת C, אנו נראה אותם כקריאות לספריית שגרות. הם אינם חלק מספרית C הסטנדרטית, אולם כפי הנראה, יהיו זמינים בכל מערכת המכילה קריאות מערכת אלו. השגרות enter_item ו-remove_item אשר איננו מראים, מטפלות ברישום של הכנסת פריטים לחוצץ והוצאת ממנו.
כעת נחזור למצב התחרות. מצב זה יכול לקרות כי הגישה ל-count אינה מבוקרת.

עמוד 41

המצב הבא יכול לקרות. החוצץ ריק והצרכן בדיוק קרא את count כדי לראות אם הוא 0. באותו רגע בקר הזמנים מחליט להפסיק את פעולת הצרכן, ולהפעיל את היצרן באופן זמני, ומתחיל להריץ את היצרן. היצרן מכניס פריט לחוצץ, ומגדיל את count, ומבחין שהמונה כרגע 1. היות והמונה היה 0, לפיכך הצרכן ישן, ולכן הוא קורא לשגרה WAKEUP כדי להעיר את הצרכן. לרוע המזל, הצרכן עדיין לא ישן, אזי איתות ההערה אובד. כאשר הצרכן מופעל שוב על ידי המעבד הוא יבחן את ערכו של המונה שהוא קרא מקודם, ימצא 0 וילך לישון. במוקדם או במאוחר היצרן ימלא את החוצץ וילך לישון גם הוא. ושניהם ישנו לעד.
עיקר הבעיה כאן היא שהערה שנשלחה לתהליך שעדיין לא ישן, אבדה. לו היא לא הייתה אובדת, הכל היה פועל כשורה. תיקון מהיר הוא לשנות את החוקים ולהוסיף WAKEUP WAITING BIT לתמונה. כאשר פקודת הערה נשלחת לתהליך שכבר ער, הביט הזה "נדלק". מאוחר יותר כאשר התהליך מנסה ללכת לישון, אם הביט הזה דלוק הוא יכבה אותו, אולם התהליך יישאר ער.
אולם, בנקל נוכל להראות דוגמאות שמשתתפים בהם שלושה או יותר תהליכים ואזי לא מספיק WAKEUP WAITING BIT אחד. נוכל לבצע שיפור נוסף ולהוסיף ביטים, אולם בעיקרון הבעיה עדיין נמצאת.

2.2.5 סמפורים

זה היה המצב בשנת 1965 כאשר דייקסטרה הציע להשתמש משתנה כדי לספור את מספר ה-WAKEUPS ולשמרו לשימוש מאוחר יותר. בהצעתו, הוצג משתנה חדש שנקרא סמפור. ערכו של הסמפור יכול להיות 0, המצביע שאף הערה (מלשון להעיר משינה) לא נשמרה, או ערך חיובי כלשהו, אם התרחשו הערות כלשהן.

דייקסטרה הציע שתי פעולות DOWN ו-UP. (הכללה של שינה והערה בהתאמה). פעולת DOWN בסמפור בודקת בכניסה לקטע הקריטי אם הערך של הסמפור חיובי. אם כן הוא מקטין את ערכו וממשיך. עם ערכו של הסמפור הוא 0 התהליך ילך לישון. בדיקת הערך, שינויו, ואפשרות להליכה לישון, כל זה נעשה כתהליך בלעדי, אקסלוסיבי, פעולה אטומית. הוא מבטיח שכאשר סמפור מתחיל לפעול אף תהליך אחר לא יוכל להכנס לסמפור עד שהפעולה הסתיימה, או נחסמה. בלעדיות זו חיונית באופן אבסולוטי לפתרון בעית הסנכרון, ולהמנעות וממצב תחרות.

פעולת UP מבוצעת ביציאה מהקטע הקריטי ומגדילה את ערכו של הסמפור ב-1. אם יש תהליכים שישנים על הסמפור, שלא השלימו את פעולת ה-DOWN שלהם, אזי באופן אקראי אחד מהם מתעורר ומשלים את הפעולה של DOWN + כניסה לקטע הקריטי.

לפיכך, אחרי שפעולת UP על סמפור, עם תהליכים ישנים עליו, הסמפור עדיין יהיה 0, אבל יהיו אז תהליך אחד פחות שישן עליו. הפעולה של הגדלת סמפור, והערת תהליך אחד היא אקסלוסיבית. אף תהליך לא יחסם בעת ביצוע UP, בדיוק כמו שאף תהליך לא יחסם בעת ביצוע WAKEUP במודל הקודם.

עמוד 42

<u>פתרון בעיית יצרן-צרכן על ידי סמפורים</u>

סמפורים פותרים את בעית איבוד ה-WAKEUP כפי שנראה באיור 2-11. חיוני שהם יבוצעו באופן אקסלוסיבי. הדרך הרגילה לבצע UP ו-DOWN כקריאות מערכת, כשמערכת ההפעלה לזמן קצר מנתקת את כל הפסיקות, בעוד היא בודקת את הסמפור, מעדכנת אותו, ומרדימה תהליך, אם צריך. היות וכל פעולות אלה משתמשות בהוראות מועטות בלבד, לא נגרם כל נזק מניתוק הפסיקות. אם משתמשים בכמה מעבדים, יש להגן על כל סמפור על ידי משתנה מנעול, עם הוראות TSL המשמשות להבטיח שמעבד אחד בלבד יבדוק את הסמפור, ברגע נתון. תהיה בטוח שהבנת שהשימוש ב-TSL כדי למנוע מכמה מעבדים להיכנס לסמפור בו זמנית שונה מהמתנה פעילה על ידי יצרן או צרכן הממתינים לחוצץ מלא או ריק. זמן פעולת הסמפור הוא כמה מיקרו שניות בלבד, ואילו היצרן והצרכן עלולים לקחת זמן שרירותי כלשהו.

הפתרון הבא משתמש בשלושה סמפורים: אחד נקרא FULL ומשמש לקריאת מספר החריצים שמלאים, אחד נקרא EMPTY ומשמש לספירת מספר החריצים הריקים, והשלישי נקרא MUTEX כדי להבטיח שיצרן וצרכן לא יכנסו לחוצץ בו זמנית. FULL מאותחל ל- 0, EMPTY מאותחל למספר החריצים שבחוצץ, ו-MUTEX . מאותחל ל-1. (סמפור מאותחל ל- 1, כלומר רק לתהליך אחד מותר להיכנס לקטע הקריטי).

סמפורים שמאותחלים ל- 1 ומשמשים על ידי שניים או יותר תהליכים להבטיח שרק אחד מהם יוכל להיכנס לקטע הקריטי שלו, ברגע נתון נקראים <u>סמפורים בינארים.</u> אם כל תהליך מבצע DOWN לפני הכניסה לקטע קריטי, ו-UP בדיוק אחרי שהוא יוצא מקטע קריטי, מניעה הדדית מובטחת.

כעת יש לנו תקשורת פנימית בין תהליכים, פשוטה וטובה, בשליטתנו, בו נלך חזרה ונתבונן בסדרת הפסיקה באיור 2-5. במערכת המשתמשת בסמפורים, הסרך הטבעית להסתרת פסיקות היא להשתמש בסמפור, מאותחל ל- 0, ומקושר עם כל התקן קלט\פלט. מייד אחרי איתחול התקן קלט\פלט, התהליך המהלך מבצע SOWN בסמפור המקושר, וגורם לחסימה מיידית. כאשר הפסיקה נכנסת, מנהל הפסיקות מבצע UP בסמפור המקושר, אשר גורם לתהליך הרלוונטי לרוץ שוב. במודל זה, שלב 5 באיור 2-5 מורכבת מביצוע UP בהתקני הסמפור, כך שבשלב 6 בקר הזמנים יוכל להפעיל את מנהל ההתקנים. כמובן, אם כמה תהליכים מוכנים כעת, בקר הזמנים עלול להפעיל תהליך חשוב יותר. אנו נתבונן כעת כיצד בקר הזמנים פועל מאוחר יותר בפרק זה.

בדוגמא באיור 2-11, ממש השתמשנו בסמפורים בשתי דרכים שונות. בסמפור MUTEX משתמשים למניעה הדדית. הוא מתוכנן כדי להבטיח שרק תהליך אחד יוכל לכתוב או לקרוא בחוצץ, ובמשתנה. המניעה ההדדית דרושה כדי למנוע תוהו ובוהו.

עמוד 43

באיור 2-11 מוצג פתרון לבעית יצרן-צרכן תוך שימוש בסמפורים.

שימוש נוסף לסמפורים הוא לסנכרון. הסמפורים FULL ו- EMPTY דרושים להבטיח מרצף אירועים מסוים, שיקרה או לא יקרה. במקרה זה, הם מבטיחים שגרה תפסיק לפעול כאשר החוצץ מלא, והצרכן מפסיק לפעול כאשר החוצץ ריק. שימוש זה שונה ממניעה הדדית.

EMPTY ביצרן מסתנכרן עם EMPTY בצרכן כך, שאם(EMPTY) DOWN מורה על 0 אזי המאגר מלא והיצרן לא ימשיך להכניס איברים, הוא יכנס לתרדמה עד אשר הצרכן יגיע לפקודת (EMPTY) UP וזה סימן הוא הוציא איבר ואז ליצרן יש מקום להכניס איבר.

FULL בצרכן מסתכרן עם FULL ביצרן כך שאם (FULL) DOWN מורה על 0 המאגר ריק והצרכן אינו יכול להמשיך להוציא איברים, הוא יכנס לתרדמה עד אשר היצרן יגיע לפקודת (FULL) UP וזה סימן שהוא הכניס איבר ואז הצרכן יכול להוציא איבר.

סמפור לסנכרון בין תהליכים – נתונים שני תהליכים המכילים בין היתר שני קטעים קריטיים. הדרישה היא למימוש תור, כלומר שקטע S1 יבוצע תמיד אחרי קטע S2. תהליך S2 לא יבוצע עד אשר תבוצע פעולת UP אשר תעיר אותו ורק אחרי שבוצע הקטע S1.

מונה אירועים

הפתרון לבעיית יצרן-צרכן הכולל שימוש בסמפורים נסמך על מניעה הדדית כדי למנוע מצב תחרות. אולם, אפשר לתכנן תוכנית לפתרון הבעיה שאינה דורשת מניעה הדדית. בפרק זה נתאר שיטה זו.

עמוד 44

השיטה משתמשת בסוג מיוחד של משתנה הנקרא מונה אירועים. 3 פעולות מוגדרות במונה אירועים E:

1. READ E: החזר את הערך של E.
2. ADVANCE E : הגדל את ערכו של E ב- 1, בפעולה אטומית (בלעדית).
3. AWAIT E,V : המתן עד של-E יש את ערכו של V או יותר.

בעית יצרן- צרכן אינה משתמשת ב-READ , אולם, יש בו צורך עבור בעיות סינכרון אחרות.

שימו לב שערכו של שמונה האירועים רק עולה, לעולם לא יורד. הם תמיד מתחילים ב- 0.

איור 2-12 מראה את פתרון בעיית יצרן-צרכן שוב, אולם הפעם בשימוש של מונה אירועים.

משתמשים בשני מוני אירועים. הראשון, IN, מונה את המספר המצטבר של הפריטים שיצרן מכניס לחוצץ, מתחילת הפעלת התוכנית.

עמוד 45

מונה האירועים השני, OUT, מונה את המספר המצטבר של פריטים שהצרכן הוציא מהחוצץ, עד עתה. ברור ש-IN חייב להיות גדול או שווה ל-OUT, אולם לא יותר מגודלו של החוצץ.

כאשר היצרן חישב פריט חדש, הוא בודק האם יש מקום בחוצץ, תוך שימוש בקריאת מערכת AWAIT. בתחילה, OUT יאותחל ל- 0 ו- SEQUENCE יאותחל למינוס N ויהיה שלילי, כך שהיצרן לא יחסם. אם היצרן מיצר +N1 פריטים לפני שהצרכן יתחיל, הצהרת ה-AWAIT תמתין עד ש-OUT יהפוך ל- 1, לפעמים זה יקרה רק אחרי שהיצרן יוציא פריט אחד. הלוגיקה של הצרכן היא פשוטה. לפני שהוא מנסה להוציא את הפריט ה-K הוא ימתין עד אשר IN יגיע ל-K, כלומר, שהיצרן יכניס K פריטים לחוצץ.

מוניטורים.

עם סמפורים ומוני אירועים, תקשורת פנימית בין תהליכים נראית פשוטה, לא כן? ממש לא. התבוננו מקרוב בפקודת DOWN לפני הכניסה או הוצאת פריט מהחוצץ באיור 2-11. נניח שנחליף את הסדר בין שני ה-DOWN בקוד של השגרה, כך ש-MUTEX יופחת לפני EMPTY, במקום אחריו. אם החוצץ היה מלא לחלוטין, השגרה תחסם, וערכו של MUTEX הוא 0. כתוצאה מכך, בפעם הבאה שצרכן ינסה להיכנס לחוצץ, הוא יבצע

DOWN ב-MUTEX, שהוא כרגע 0, וגם חסום. שני התהליכים יישארו חסומים לנצח, ולא יבצעו כל פעולה נוספת. מצב ביש זה נקרא DEADLOCK.
במילים אחרות, השימוש בסמפור נתון להחלטת המשתמש. מספיק שהמשתמש יחליף את סדר הופעת ה- DOWN וה-MUTEX ושני התהליכים היו נכנסים לחסימה. המשתמש חייב מאוד להזהר כאשר הוא משתמש בסמפור. על מנת להקל על כותבי התוכניות פיתחו מנגנון ברמה גבוהה יותר של סנכרון- פרימיטיבי, הקרוי מבני פיקוח – MONITOR.
MONITOR הוא אוסף של שגרות, משתנים ומבני נתונים המקובצים יחדיו במודל מיוחד או מארז. תהליכים יכולים לקרא לשגרות במוניטור מתי שירצו, אולם אינם יכולים להיכנס ישירות למבני הנתונים הפנימי של המוניטור מפרוצדורות שמוגדרות מחוץ למוניטור. איור 2-13 מדגים מוניטור הכתוב בשפה דימיונית, דמוית פסקל.
מוניטורים יעילים בהשגת מניעה הדדית: רק תהליך אחד יכול להיות פעיל בתוך המוניטור בזמן נתון. המוניטור מתוכנת באופן מיוחד כך שהקומפיילר מכיר זאת ויכול לנהל קריאות לשגרות שבתוכו באופן שונה משאר הקריאות. כאשר תהליך קורא לפרוצדורות במוניטור, ההוראות הראשונות שרצות בודקות
האם יש תהליך אחר שפעיל, אם כן התהליך מושהה עד שהתהליך הפעיל יעזוב את המוניטור, אחרת, הוא יכנס ויהיה פעיל. על הקומפיילר לבצע את המניעה ההדדית בכניסות למוניטור בדרך כלל על ידי שימוש בסמפור בינארי. היות והקומפיילר, ולא המשתמש, מנהל את ביצוע המניעה ההדדית, יש פחות סיכוי שמשהו ישתבש.

עמוד 46

בכל מקרה, האדם שכותב למוניטור לא צריך לדעת איך הקומפיילר מנהל את המניעה ההדדית. מספיק לדעת שעל ידי הפנית כל הקטעים הקריטיים אל המוניטור, שני תהליכים לעולם לא יכנסו לקטע הקריטי שלהם בו זמנית.
אף על פי שמוניטור מספק דרך קלה להשגת מניעה הדדית, זה לא מספיק. אנו צריכים דרך לחסום תהליכים כאשר אינם יכולים להמשיך. בבעיית יצרן-צרכן, קל מאוד לבצע את כל בדיקות תכולת החוצץ (מלא או ריק) על ידי שגרות מוניטור. אולם למשל, איך היצרן יחסם כאשר הוא מוצא את החוצץ מלא?
כלומר למרות שהמוניטור משיג את התנאי של מניעה הדדית, הוא חייב לממש גם את התנאי של החסימה. כלומר, תהליך חייב לחסום את עצמו כאשר אין הוא יכול להמשיך וכך לאפשר לתהליך אחר להיכנס לקטע הקריטי. (לדוגמא, בבעיית היצרן-צרכן, כאשר המאגר מלא היצרן צריך לחסום את עצמו עד אשר הצרכן יוציא לפחות איבר אחד).
הפתרון לכך הוא בהוראות הנקראות CONDITION VARIABLES, ובשתי פעולות עליהם, WAIT ו-SIGNAL כאשר שגרת מוניטור מגלה שאינה יכולה להמשיך (היצרן מוצא את החוצץ מלא), אזי הוא מבצע WAIT על כמה משתני תנאי, למשל FULL. פעולה זו גורמת לתהליך הקורא להיחסם. כמו כן הוא מתיר לתהליך אחר שהמתין לכניסה למוניטור להיכנס כעת.
התהליך השני לדוגמא, הצרכן, יכול לעורר משנתו את תהליך היצרן על ידי ביצוע SIGNAL על משתנה התנאי שהיצרן ממתין לו (כלומר התנאי שבגללו היצרן נרדם מפסיק להתקיים והוא יכול לפעול שוב). כדי להימנע משני תהליכים פעילים במוניטור בו זמנית אנו זקוקים לחוק המגדיר מה קורה לאחר SIGNAL. הוצע לתת לתהליך שהוער פרק זמן לפעולה, תוך השעיית שאר התהליכים. הצעה נוספת הייתה שתהליך המבצע סיגנל חייב לצאת מהמוניטור מייד. במילים אחרות, הצהרת סיגנל חייבת להופיע רק בסוף שגרת המוניטור. אנו נקבל את ההצעה הפשוטה יותר האומרת, אם סיגנל בוצע על משתנה תנאי אשר כמה תהליכים ממתינים לו, רק אחד מהם, הנבחר על ידי המערכת, יתעורר.

עמוד 47

משתני תנאי אינם מונים. הם אינם צוברים סיגנלים לשימוש מאוחר יותר, כפי שעושים סמפורים. לפיכך, אם מבוצע סיגנל על משתנה תנאי, כאשר אף אחד לא ממתין לתנאי, הסיגנל אובד. הוראת WAIT מוכרחה לבוא לפני הסיגנל אחרת הסיגנל ילך לאיבוד.
החוק הזה מפשט את הביצוע. למעשה זו לא בעיה כי ניתן לעקוב אחרי מצב כל תהליך עם משתנים, אם צריך. תהליך שמבצע סיגנל יכול לראות שפעולה זו אינה נחוצה על ידי התבוננות במשתנה.
באיור 2-14 מודגם תמצית לפתרון בעיית יצרן-צרכן תוך שימוש במוניטורים בדמוי פסקל.

ניתן לחשוב שפעולות WAIT ו-SIGNALזהות לפעולות SLEEP ו-WAKEUP , שכפי שראינו מביאות למצב תחרות פאטלי. הם מאוד דומות, אולם עם הבדל משמעותי אחד: SLEEP ו-WAKEUP נכשלו כאשר תהליך אחד ניסה ללכת לישון, והשני ניסה להעיר אותו. עם מוניטורים, הדבר לא יכול לקרות. המניעה ההדדית האוטומטית בשגרות המוניטור מבטיחות שאם, נאמר, היצרן בתוך שגרת המוניטור מגלה שהחוצץ מלא, הוא יוכל להשלים את פעולת WAIT מבלי לדאוג לאפשרות שבקר הזמנים יפסיק את פעולתו ויפעיל את הצרכן בדיוק לפני השלמת פעולת WAIT. הצרכן לא יוכל אפילו להיכנס למוניטור עד אשר פעולת WAIT תסתיים והשגרה תסומן כלא פועלת יותר.
על ידי הפיכת המניעה ההדדית באזורים קריטיים לפעולה אטומית, מוניטורים המבצעים תכנות מקבילי פחות מועדים לפורענות מאשר עם סמפורים. אולם עדיין גם להם יש חסרונות. לא בכדי איור 2-14 נכתב בשפה דמוית פסקל ולא ב-C בדומה לדוגמאות האחרות בספר. כפי שציינו קודם, מוניטורים כתובים בצורה מיוחדת כך שהקומפיילר יוכל לזהותם, ולנהל מניעה הדדית בכניסות אליהם. ברוב השפות אין מוניטורים ולכן לא הגיוני לצפות מקומפיילרים שלהם לאכוף חוקי מניעה הדדית. למעשה, איך יכול קומפיילר בכלל לדעת איזה מהשגרות שייכת למוניטור ואיזה לא?
לשפות אלו אין גם סמפורים, אולם קל להוסיף סמפור: כל שיש לעשות הוא להוסיף שתי שגרות הכתובות בשפת אסמבלי לספריה, לביצוע קריאות המערכת UP ו-DOWN. הקומפיילר אפילו לא צריך לדעת שהן קיימות. כמובן שמערכות ההפעלה צריכות לדעת על הסמפורים, אולם לפחות אם החלטת לכתוב סמפורים המתבססים על מערכת ההפעלה, אתה יכול לכתוב את תוכניות המשתמש עבורן ב-C או פסקל או אפילו ביסיק אם אתה מזוכיסטי דייך. עם מוניטורים, אתה זקוק לשפה שמכילה אותם. כלומר, חסרון המוניטור הוא שמעט שפות בלבד מוניטור מובנה בתוכן.
למוניטורים ולסמפורים בעיה נוספת, והיא שהם תוכננו לפתור את בעית המניעה ההדדית במחשב עם מעבד אחד או יותר שלכולם גישה לזיכרון משותף. כלומר אם נשים סמפורים (או אפילו מונים) בזיכרון משותף ונגן עליהם עם TSL, נוכל להמנע ממצב תחרות. אולם במערכות מבוזרות המכילות כמה מעבדים אשר לכל אחד שטח זיכרון משלו, אשר מחוברים על ידי רשת מקומית, או החלפת מידע בין מכונות שונות, סמפורים לא מתאימים. המסקנה היא שסמפורים הם ברמה נמוכה מידי, ומוניטורים אינם שימושיים למעט שפות תכנות מועטות. ולכן זקוקים לפתרון אחר.

עמוד 48

פתרון מוניטור לבעית היצרן-צרכן.

עמוד 49

2.2.8 העברת הודעות MESSAGE PASSING

הפתרון האחר הוא העברת הודעות. שיטה זו של תקשורת פנימית בין תהליכים משתמשת ב- SEND וב-RECEIVE, אשר בדומה לסמפורים, ולא בדומה למוניטורים הם קריאות מערכת, במקום מבנים בשפה. ככאלו אנו יכולים בקלות להכניסם לספרית שגרות כגון

SEND (DESTINATION, &MESSAGE)

RECEIVE (SOURCE, & MESSAGE)

הראשון שולח הודעה ליעד נתון, והשני מקבל הודעה ממקור נתון. אם אין אף הודעה המקלט יכול להיחסם עד שמתקבלת הודעה.

נושאי תכנון עבור מערכת של העברת הודעות

להודעות העוברות במערכת ישנם הרבה בעיות שיש בהן אתגר, ובעיות תכנון שמתעוררות עם סמפורים או מוניטורים, במיוחד אם התהליכים המקושרים הם במחשבים שונים המקושרים ברשת. לדוגמא, הודעות יכולות ללכת לאיבוד ברשת. כדי לשמור שהודעות לא יאבדו, השולח והמקבל יכולים להסכים שכשההודעה מתקבלת, המקבל ישלח חזרה הודעת אישור מיוחדת. אם השולח לא קיבל את הודעת האישור, בזמן קבוע כלשהו, הוא ישלח את ההודעה שוב.
כעת נניח שההודעה שנשלחה הגיעה בשלמותה ליעד, אולם הודעת האישור אבדה. השולח ישלח שוב את ההודעה, והמקבל יקבל אותה פעמיים. חיוני אז שהמקבל יוכל להבחין בין הודעה חדשה לבין ישנה. בדרך כלל בעיה זו נפתרת על ידי מספור ההודעות במספרים

רציפים. אם המקבל מקבל הודעה אם אותו רצף מספרים כמו בהודעה הקודמת, הוא יודע שזהו העתק, ויש להתעלם ממנה.
מערכת הודעות צריכה להתמודד עם שאלת שמות התהליכים, כך שתהליך יצוין בקריאת SEND או RECEIVE בברור, ודו משמעות תמנע. לעיתים קרובות משתמשים בשמות כגון process@machine או machine:process.
אם מספר המחשבים גדול מאוד, ואין שליטה על מתן השמות למחשבים, יכול לקרות ששני ארגונים יקראו למחשבים שלהם בשם זהה. בעיה של דו משמעות יכולה להתעורר. בעיה זו נפתרת על ידי חלוקת המחשבים לשטחי פעולה מזהים (domain) ולקרוא למחשבים process@machine.domain. כך לא תתעורר בעיה אם לשני מחשבים יש שם זהה, בתנאי שהם נמצאים בתחום שונה.
כמו כן שמות האזורים צריכים להיות מיוחדים.
זיהוי השולח הוא נושא חשוב במערכת הודעות: איך יכול לקוח לדעת שהוא מקושר לשרת שהוא צריך להיות מקושר אליו ולא למתחזה? איך יכול שרת לדעת איזה לקוח ביקש את הבקשה? הפתרון של הצמדת קוד ייחודי להודעה הידוע רק למשתמש מורשה, יכול לעזור. מצד שני ישנם נושאים חשובים לתכנון כאשר השולח והמקבל נמצאים באותו מחשב.

עמוד 50

אחד מהם הוא זמן ביצוע. העתקת הודעות מתהליך אחד למשנהו תמיד איטית יותר, מאשר לבצע פעולת סמפור, או להיכנס למוניטור. הרבה עבודה הושקעה בייעול העברת הודעות. קרייטון למשל הציע הגבלה על גודל ההודעות לגודל שיתאים לאוגרים, ואזי להשתמש באוגרים להעברת הודעות.

בעיית היצרן-צרכן ומערכת שליחת הודעות

נציג את פתרון בעיית היצרן-צרכן על ידי שימוש בהעברת הודעות ולא בשטח זיכרון משותף. הפתרון מוצג באיור 2-15. אנו מניחים שכל ההודעות הם בגודל שווה, ושהודעה שנשלחה ועדיין לא הגיעה ליעדה מאוחסנת בחוצץ באופן אוטומטי על ידי מערכת ההפעלה. בפתרון זה, ניתן להעביר N הודעות (אנלוגיה ל-N חריצים שהיו בחוצץ הזיכרון המשותף. הצרכן מתחיל על ידי שליחת N הודעות ריקות ליצרן. בכל פעם שליצרן יש פריט להעביר לצרכן, הוא לוקח הודעה ריקה ושולח אותה חזרה מלאה. בדרך זו, סך כל ההודעות נשאר קבוע כל הזמן כך שניתן לאחסנם בגודל מסוים של זיכרון.
אם היצרן פועל מהר יותר מהצרכן, כל ההודעות יסתיימו כמלאות וממתינות לצרכן. היצרן יחסם כשהוא ממתין להודעה ריקה שתגיע. אם הצרכן פועל מהר יותר, אזי קורה ההפך: כל ההודעות יהיו ריקות וימתינו ליצרן שימלא אותן. אזי יחסם הצרכן עד כשהוא ממתין להודעה מלאה.
גרסאות רבות אפשריות עם העברת הודעות. בתחילה נתבונן כיצד הודעות ממוענות. דרך אחת היא להקצות לכל תהליך כתובת מסוימת ונמען את ההודעות לכתובת התהליך. דרך אחרת היא להמציא מבנה נתונים חדש, שניקרא MAILBOX. תא דואר הוא מקום לאחסון מספר מסוים של הודעות. כאשר משתמשים בתא דואר פרמטר הכתובת בקריאות-SEND וב-RECEIVE הם תאי דואר, ולא תהליכים. כאשר תהליך מנסה לשלוח הודעה לתיבת דואר מלאה, הוא מושעה עד שתוצא הודעה מתא הדואר.
עבור בעיית היצרן צרכן, שניהם, גם היצרן וגם הצרכן יצרו תיבות דואר גדולות מספיק להכיל N הודעות. היצרן ישלח הודעות המכילות נתונים לתא הדואר של הצרכן, והצרכן ישלח הודעות ריקות לתא הדואר של היצרן. כאשר משתמשים בתא דואר, תהליך האחסון ברור: תא היעד מאחסן הודעות שנשלחו לתהליך היעד אולם טרם נתקבלו על ידו.
הקיצוניות השניה מהקצאת תיבות דואר היא הימנעות מכל אחסנה. כאשר גישה זו מיושמת, אם SEND מבוצע לפני RECEIVE, התהליך השולח נחסם עד ש- RECEIVE מתרחש, בכל פעם ניתן להעתיק הודעה ישירות מהשולח למקבל, ללא תווך של חוצץ.
בדומה אם RECEIVE מבוצע קודם, המקבל נחסם עד ש-SEND מתרחש. אסטרטגיה זו נקראת מפגש (RENDEZVOUS). והוא קל יותר לביצוע מאשר שיטת אחסון ההודעות, אולם פחות יציבה היות והשולח והמקבל מוכרחים לפעול יחד.
התקשורת הפנימית בין תהליכים ביוניקס היא דרך קריאת המערכת PIPE (צינור) שהם תאי דואר יעילים.

עמוד 51

ההבדל היחיד בין מערכת הודעות עם תאי דואר לבין השימוש בצינור הוא שצינורות לא שומרים על גבולות בין הודעות. במילים אחרות אם תהליך כתב 10 הודעות של 100 בתים לצינור ותהליך אחר קרא 1000 בתים מאותו צינור, הקורא יקבל את כל 10 ההודעות בבת אחת. עם מערכת הודעות אמיתית, כל READ יחזיר רק הודעה אחת. כמובן, אם תהליך מסכים תמיד לקרוא ולכתוב גודל קבוע של הודעות מהצינור, או לסיים כל הודעה בתו מיוחד, לא תתעורר כל בעיה.

באיור 2-15 פתרון בעיית היצרן-צרכן עם N הודעות.

2.2.9 לא כלול בחומר הלימוד

עמוד 56

2.3 בעיות IPC קלאסיות

הבעיות המתוארות בפרק זה הן בעיות קלאסיות בתחום שיתוף הפעולה בין תהליכים.

2.3.1 בעיית הפילוסופים הסועדים.

בשנת 1965 דייקסטרה הציג ופתר בעיית סנכרון הנקראת בעיית הפילוסופים הסועדים. מאז כל מי שממציא פתרון סנכרון כלשהו מנסה להדגים כמה נהדר הפתרון הזה, וכמה אלגנטי הוא פותר את בעיית הפילוסופים הסועדים. להלן תאור הבעיה: 5 פילוסופים יושבים מסביב לשולחן עגול. לכל פילוסוף צלחת של ספגטי. הספגטי כל כך חלקים שהפילוסופים צריכים 2 מזלגות כדי לאכול. בין כל 2 צלחות יש מזלג. השולחן מצויר באיור 2-18. חיי סועד מורכבים מזמן בו הוא אוכל וזמן בו הוא חושב.

עמוד 57

כאשר פילוסוף נהיה רעב הוא מנסה לקחת את שני המזלגות קודם השמאלי ואחר כך הימני, (בסדר זה) ולאכול. אם מצליח הוא אוכל, ואח"כ מניח אותם וממשיך לחשוב, אחרת, הוא תקוע, אינו יכול להמשיך לאכול. שאלת המפתח היא האם אתה יכול לכתוב תוכנית כך שאף סועד לא יתקע? איור 2-19 מראה פתרון פשוט. השגרה take_fork ממתינה עד אשר מזלג מסוים פנוי ואז לוקחת אותו. לדאבוננו פתרון זה הוא שגוי. נניח שכל 5 הסועדים ייקחו את המזלג הנמצא משמאלם, ביחד. אף אחד מהם לא יוכל לקחת את המזלג הימני שלו, ואז יווצר DEADLOCK (מצב של קיפאון – אף סועד לא יכול לאכול למרות שהוא מנסה).

אנו נשנה את התוכנית כך שאחרי שסועד לקח את המזלג השמאלי, התוכנית בודקת האם המזלג הימני פנוי. אם לא הפילוסוף מניח חזרה את השמאלי, ממתין זמן מה, ואזי חוזר על התהליך. הצעה זו שגויה גם כן, מסיבה שונה. נניח שכל הפילוסופים החלו את התהליך בו-זמנית, לקיחת המזלג השמאלי, בדיקת הימני, הנחת השמאלי והמתנה לזמן מסוים, וכן הלאה, לעולמי-עד. מצב זה נקרא STARVATION (הרעבה – מצב בו תהליך אינו יכול להתקדם בגלל סיבות שאין לו קשר אליהן. לדוגמא מצב של קיפאון מתמיד, או שהמעבד לא מקצה זמן לתהליך, או סועד שאינו יכול לאיכול משום שתמיד שכניו אוכלים).

כעת אנו עלולים לחשוב "אם כל פילוסוף ימתין זמן אקראי במקום זמן קבוע מראש לכולם, אחרי כישלון בהשגת המזלג הימני, הסיכוי להיתקע למשך שעה, למשל, הוא קטן מאוד. השקפה זו היא נכונה, אולם כמה יישומים יעדיפו פתרון שתמיד עובד ולא נכשל למשך סדרה לא צפויה של מספרים אקראיים. (תחשבו על מערכת אבטחה בכור אטומי). שיפור לתוכנית באיור 2-19 שאין בה קיפאון ולא הרעבה, הוא להגן על 5 ההצהרות שבאות אחרי הקריאה ל-THINK על ידי סמפור בינארי. לפני הבקשה ללקיחת מזלגות, הפלוסוף מבצע DOWN ב-mutex. אחרי האכילה והחזרת המזלגות הוא יבצע UP ב-mutex.

עמוד 58

באופן תאורטי פתרון זה מספק. באופן מעשי יש בו "באג": רק פילוסוף אחד יכול לאכול בזמן מסוים. עם 5 מזלגות פנויים, אנו צרכים להיות מסוגלים להרשות לשני פילוסופים לאכול בו זמנית.

הפתרון שמוצג באיור 2-20 הוא נכון, ומאפשר למקסימום מקביליות, למספר שרירותי של סועדים. הוא משתמש במערך, STATE, לשמור מעקב על מצבו של הפילוסוף, האם אוכל, חושב, או רעב (מנסה להשיג מזלג). פילוסוף יכול לעבור למצב אכילה אם אף אחד

משכניו לא אוכל. שכני הפילוסוף (i) מוגדרים במקרו LEFT ו-RIGHT. במילים אחרות, אם (I) הוא 2 אזי LEFT הוא 1 ו-RIGHT הוא 3.
התוכנית משתמשת במערך של סמפורים, אחד לכל פילוסוף, כך שפילוסוף רעב יוכל להיחסם אם המזלגות הדרוש לא פנויים. נעיר שכל תהליך מפעיל את השגרה philosopher כקוד ראשי, אולם השגרות האחרות, take forks, put_forks ו-test הן שגרות רגילות ולא תהליך נפרד.

- שימוש בעיית הפילוסופים הסועדים – בעיה זו שימושית להדגים תהליכים שרוצים גישה למספר מוגבל של משאבים, כמו אמצעי קלט\פלט.

2.3.2 בעיית הקוראים כותבים

בעיה מפורסמת אחרת היא בעיית הקוראים כותבים, המדגימה גישה למאגר נתונים משותף. תארו לעצמכם מאגר נתונים ענק כמו של מערכת הזמנות לחברת תעופה, עם הרבה תהליכים מתחרים הרוצים לכתוב ולקרוא ממנו. ניתן שכמה תהליכים יקראו את מאגר הנתונים בו זמנית, אולם אם תהליך אחד כותב במאגר הנתונים, לשום תהליך אחר לא תינתן גישה למאגר, אפילו לא לקריאה. השאלה היא כיצד תתכנת את בעיית קוראים-כותבים? פתרון אחד מודגם באיור 2-21. בפתרון זה, הקורא הראשון שנכנס למאגר הנתונים מבצע DOWN על הסמפור db. קוראים מאוחרים יותר רק יגדילו את המונה, rc. כאשר הקוראים עוזבים את המאגר לאחר הקריאה, הם יקטינו את המונה, והאחרון יבצע UP על הסמפורים, מאפשר לממתין לכתיבה החסום כרגע, אם ישנו, להיכנס למאגר.
נרמז בפתרון זה שלקוראים יש עדיפות על הכותבים. אם כותב מופיע, בעוד כמה קוראים נמצאים במאגר הנתונים, הכותב חייב לחכות. אם קוראים חדשים מצטרפים, כך שתמיד תהיה רשימה של קוראים במאגר, הכותב יאלץ להמתין עד אשר אף קורא לא יבקש להיכנס למאגר. פתרון שהוצע הוא לתת עדיפות לכותב.

עמוד 59

איור 2-21 ובו הפתרון לבעיית קוראים כותבים. בפתרון 2 סמפורים:

1. mutex – מאפשר קריאה במקביל על ידי כמה קוראים.
2. Db (data-base) – מגן על כניסת תהליכים לקטע הקריטי במקביל.

הקורא הראשון מבצע פעולת DOWN לסמפור db, כאשר מגיעים קוראים נוספים הם מעלים את המונה של rc (מונה הקוראים), וכאשר קוראים עוזבים הם מפחיתים ממונה זה עד שהאחרון יבצע up לסמפור db ויאפשר לכותב חסום אם יש כזה להכנס.

בפרוצדורת READER הסמפור mutexמגן על משתנה מונה מספר קוראים rc מפני כתיבה בו (עדכון מספר הקוראים) בו זמנית מכיוון שכמה קוראים יכולים להריץ שגרה זו בו-זמנית. אם rc=1 כלומר הקורא הראשון אזי מתבצע DOWN לסמפור db שיגן מפני כתיבה בזמן קריאה.
בפרוצדורת WRITER: כותב מבצע DOWN לסמפור db, אך הוא יאלץ להמתין עד אשר יבוצע UP ל-db עד אחרון הממתינים בתור לכניסה לקטע הקריטי.
הבעיה בפתרון זה היא שאם כל הזמן קוראים נכנסים הם לא יאפשרו לכותב שממתין להיכנס למצב של כתיבה – הרעבת הכותב.

בעיית הספר הישן

בעיית IPC קלאסית נוספת מתרחשת במספרה. במספרה ספר אחד, כסא להסתפר, אחד, ו-N כסאות ללקוחות ממתינים, אם ישנם. אם אין אף לקוח, הספר יושב בכיסא הספר, ונרדם. כפי שמודגם באיור 2-22. כאשר מגיע לקוח, עליו להעיר את הספר הישן. אם לקוח נוסף מגיע, בעוד הספר מספר, הוא יכול לשבת (אם יש כסאות ריקים) או לעזוב את המספרה (אם כל הכיסאות מלאים).

עמוד 60

הבעיה היא לתכנת את הבעיה מבלי להיכנס למצב תחרות (מצב של שני לקוחות המסתפרים בו זמנית).
הפתרון שלנו משתמש בשלושה סמפורים: customers, מנהל תור של לקוחות הממתינים לתספורת ואשר מונה את מספר הלקוחות הממתינים (לא כולל הלקוח שמסתפר בכסא

הספר, אשר אינו ממתין), barbers, המתנה ללקוחות (0 – מספר כרגע, או 1 – פנוי) ו-mutex, מגן על הקטעים הקריטיים. אנו גם זקוקים למשתנה, waiting, אשר גם הוא מונה לקוחות ממתינים. חיוני שיהיה העתק של customers. הסיבה לכך היא שאין דרך לקרוא מצב נוכחי של סמפור, ובפתרון זה, לקוח שנכנס למספרה צריך לספור את מספר הלקוחות הממתינים, אם זה פחות ממספר הכיסאות הוא ממתין אחרת הוא עוזב. הפתרון שלנו מודגם באיור 2-23. כאשר הספר מופיע בבוקר לעבודה הוא מבצע את פרוצדורת barber, הגורם לו להיחסם בסמפור customer, עד שמישהו מופיע. אזי הוא הולך לישון כפי שמודגם באיור 2-22. כאשר הלקוח הראשון מופיע, הוא מפעיל את customer, המתחיל על ידי השגת mutex כדי להיכנס לקטע הקריטי. אם לקוח נוסף נכנס, זמן קצר אח"כ השני לא מסוגל לבצע דבר עד אשר mutex משוחרר.

עמוד 61

הלקוח אזי בודק האם מספר הלקוחות הממתינים קטן ממספר הכיסאות. אם לא הוא משחרר את mutex ועוזב ללא תספורת. אם יש כסא פנוי , הלקוח מגדיל את המשתנה waiting. אזי הוא מפעיל את הסמפור customers ומעיר את הספר. בנקודה זו הספר והלקוח שניהם ערים, כאשר הלקוח משחרר את metex, הספר לוקח אותו, מבצע מספר עבודות בית, ומתחיל בתספורת.
עם סיום התספורת, הלקוח יוצא מהפרוצדורה ועוזב את החנות.
שלא בדומה לדוגמא הקודמת שלנו, אין לולאה עבור הלקוח, היות ולהסתפר היא פעולה חד פעמית. לעומת זאת יש לולאה לספר, המנסה להגיע ללקוח הבא, אם הוא נוכח, ותספורת נוספת ניתנת. אם לא הספר הולך לישון.

2.4 תזמון תהליכים

בדוגמאות הקודמות היו לנו לעיתים קרובות מצבים ששני תהליכים או יותר פועלים (יצרן-צרכן). כאשר יותר מתהליך אחד פועל, מערכת ההפעלה חייבת להחליט, איזה תהליך ירוץ ראשון.

עמוד 62

החלק במערכת ההפעלה שמחליט החלטה זו קרוי SCHEDULER , והאלגוריתם לפיו הוא מחליט נקרא SCHEDULING ALGORITHM.
במערכות שיתוף בזמנים רבות משתמשים, לעיתים קרובות משולבות בעבודות BATCH ברקע, דבר הגורם לאלגוריתם התזמון להסתבך יותר. באופן קבוע ישנם משתמשים הממתינים לשרות, וישנם עבודות BATCH או אחרות הממתינות לביצוע. אפילו במערכות שיתוף זמנים טהורות יש עבודות ברקע, כמו מערכת דואר אלקטרוני, אשר בדרך כלל פועלת כל הזמן, שולחת או מקבלת דואר או חדשות מהרשת.
לפני שנדון באופן ספציפי באלגוריתם לתזמון, נחשוב מה המתזמן מנסה להשיג. לאחר הכל המתזמן לדאוג למדיניות, ולא לספק מכניזם.

עמוד 63

ישנם קריטריונים שונים לאלגוריתם תזמון טוב .

1. הוגנות FAIRNESS – האלגוריתם צריך לדאוג לכך שכל תהליך יקבל את החלק "המגיע לו" מהמעבד.
2. יעילות EFFICIENCY – האלגוריתם צריך לדאוג לכך שהמעבד יהיה עסוק 100% מהזמן.
3. זמן תגובה RESPONSE TIME- להביא למינימום את זמן התגובה לתוכניות אינטראקטיביות .
4. תפנית TURNAROUND- להביא למינימום את זמן ההמתנה של משתמשים לפלט של תוכניות BATCH.
5. תפוקה THROUGHPUT – למקסם את מספר התהליכים שרצים ביחידת זמן של שעה.

עיון קל יראה לנו שכמה ממטרות אלו סותרות. כדי לצמצם זמן התגובה עבור משתמשים אינטראקטיבים, המתזמן (SCHDULER) לא יוכל לפעול על עבודות אצווה (BATCH) בכלל (למעט אולי בין 3 ל-6 בבוקר כאשר כל המשתמשים האינטראקטיבים ישנים).

משתמשי אצווה, סביר להניח לא יאהבו את האלגוריתם הזה. בכל מקרה הוא פוגע בקריטריון מספר 4. ניתן להראות שכל אלגוריתם לתזמון, שמעדיף קבוצה של פעולות, פוגע בקבוצה אחרת. בסך הכל סך הזמן הפנוי של המעבד הוא סופי. כדי לתת למשתמש אחד יותר, תאלץ לתת לאחר פחות. אלו החיים.
המורכבות שהמתזמן צריך להתמודד אתה היא שכל תהליך הוא מיוחד, ובלתי צפוי. כמה מבזבזים זמן רב בהמתנה לקובצי קלט\פלט, בעוד אחרים ישתמשו במעבד במשך שעות, אם תהיה להם הזדמנות. כאשר המתזמן מתחיל להפעיל תהליך כלשהו, אף פעם הוא לא יודע בביטחון, כמה זמן זה ייקח עד אשר התהליך הזה יחסם, אם בגלל קלט\פלט, או סמפור, או כל סיבה אחרת. כדי להבטיח שאף תהליך לא ירוץ זמן ארוך מידי, כמעט לכל המחשבים יש שעון אלקטרוני מובנה בתוכם, אשר גורם לפסיקות, כל כמה זמן. תדירות של 50 או 60 פעמים בדקה היא נפוצה, אולם במחשבים רבים, מערכת ההפעלה יכולה לכוון את תכיפות פסיקות השעון, לפי רצונה. בכל פסיקת שעון, מערכת ההפעלה יכולה להחליט האם לאפשר לתהליך שפועל כעת, להמשיך, או שמא הוא השתמש דיו בזמן מעבד, כרגע, ויש להשעותו עבור תהליך אחר שממתין למעבד. גישה זו שמשעה תהליכים פועלים, באופן זמני נקראת
PREEMPTIVE SCHEDULING, והיא מנוגדת לאסטרטגיה מוקדמת של RUN TO COMPLETION של מערכות אצווה מוקדמות. ריצה עד קומפילציה נקראת גם NONPREEMPTIVE SCHEDULING. לפי אסטרטגיה זו התוכניות היו נכנסות לתור לפי סדר הגעתן, וכל תוכנית שהיתה בראש התור היתה מורצת מתחילתה ועד סופה – ללא מעבר בין תהליכים (עומד בדרישות של הוגנות, אולם שאר הדרישות אינן מתקיימות). אסטרטגיה זו של מערכות אצווה אינה שימושית. כפי שראינו בפרק זה, ניתן להשעות תהליך באופן שרירותי, ללא אזהרה, כדי לאפשר לתהליך אחר לרוץ. הדבר מוביל למצב תחרות, ומצריך סמפורים, מוני ארועים, מוניטורים, הודעות, או כל גישה מתוחכמת אחרת למניעת מצב התחרות.
לפיכך למרות שהאלגוריתמים של NONPREEMPTIVE SCHEFULING פשוטים וקלים
לביצוע, בדרך כלל אינם מתאימים למערכות כלליות רבות משתמשים. מאידך למערכות המוקדשות לנושא אחד כמו מאגרי נתונים, יהיה זה הגיוני לתהליך אב לאתחל תהליך בן ולתת לו לפעול עד שהוא מסיים. ההבדל בין מערכות כלליות הוא שכל התהליכים במאגרי נתונים נתונים לפיקוח של מערכת בקרה יחידה, אשר יודעת מה מבצע כל תהליך בן, ולכמה זמן.

עמוד 64

2.4.1 ROUND ROBIN SCHEFULING

כעת נתבונן בכמה אלגוריתמים ספציפים לתזמון. אחד מהישנים, פשוט, הוגן ונמצא בשימוש נרחב הוא אלגוריתם ROUND ROBIN. לכל תהליך מוקצה פסק זמן שבו הוא יכול לרוץ. זמן זה קרוי QUANTUM. אם התהליך עדיין לא הסתיים בסיום זמן זה, המעבד משעה אותו ותהליך אחר מקבל זמן ריצה. תהליך שנחסם, או סיים לפני תום הזמן המוקצב, מוחלף בתהליך אחר באופן מיידי.
ROUND ROBIN הוא אלגוריתם קל לביצוע. כל שעל המתזמן לעשות הוא לנהל תור של תהליכים הממתינים לריצה. תהליך שהופסק ועדיין לא הסתיימה ריצתו, מועבר לסוף התור והתהליך שבראש התור נכנס לריצה. ראה איור 24-2. כאשר תהליך נכנס למצב של BLOCKED הוא נכנס לתור נפרד של תהליכים חסומים עד אשר הוא משתחרר ממצב זה.
הנושא המעניין היחידי באלגוריתם זה הוא אורך הזמן המוקצב לריצה. מעבר מתהליך אחד למשנהו דורש זמן עבור אדמיניסטרציה - שמירת וטעינת רגיסטרים ומפות זיכרון, עדכון טבלאות ותורים וכדומה.
נניח שהחלפת התהליך או כפי שהיא נקראת לפעמים, CONTEXT SWICH, אורכת 5 מילישניות. כמו כן נניח שהזמן המוקצב לריצת תהליך הוא 20 מילישניות. אם פרמטרים אלו, אחרי ביצוע של 20 מילישניות של עבודה פוריה, המעבד יאלץ לבזבז 5 מילישניות על החלפת תהליכים. עשרים אחוז מזמן המעבד ילך לאיבוד על אדמיניסטרציה.

כדי לשפר את יעילות המעבד נוכל להקציב נניח 500 מילישניות לפרק זמן ריצה. עכשיו ביזבוז הזמן הוא פחות מאחוז אחד. אולם מה יקרה אם עשרה משתמשים אינטראקטיבים יכו על מקש ENTER בו זמנית. עשרה תהליכים יצטרפו לתור התהליכים שמבקשים לרוץ. אם המעבד הוא אידאלי, הראשון ירוץ מייד, השני יתכן שלא ירוץ עד, נאמר חצי שניה, וכן הלאה. האחרון, ביש המזל, יתכן שימתין 5 שניות לפני שיהיה לו סיכוי, בהנחה שכל התהליכים הקודמים ניצלו את כל פרק הזמן המוקצב להם לריצה. רוב המשתמשים יחשבו שזמן תגובה של -5 שניות לפקודה קצרה הוא נורא ואיום. את המסקנה ניתן לנסח כך: בתהליכים אינטראקטיבים נרצה קצובת זמן קצרה כדי שתהיה החלפה מהירה בין תהליכים אולם הקצבת זמן קצרה מידי תגרום להרבה החלפות בין תהליכים, ותוריד את יעילות המעבד. מאידך הקצבת זמן ארוכה מידי תגרום לזמן תגובה ארוך, לבקשות אינטראקטיביות קצרות. לכן קצובת זמן של 100 מילישניות מקובלת בדרך כלל, ונחשבת לפשרה.

עמוד 65

PRIORITY SCHEDULING 2.4.2

תזמון ROUND ROBIN מניח שכל התהליכים שווים בחשיבותם. לעיתים קרובות, לתהליכים שונים יש סדר עדיפויות שונה, והצורך לקחת זאת בחשבון, בעת הקצאת זמן מעבד, מובילה לאלגוריתם זה. והרעיון הבסיסי הוא כזה: לכל תהליך יש דרגת עדיפות, כאשר התהליך בעל העדיפות הגבוהה ביותר ירוץ ראשון.

כדי למנוע מצב שבו תהליך עם עדיפות גבוהה ירוץ לעד, המתזמן (SCHEDULER) יפחית מהעדיפות שלו בכל פעם שתופעל פסיקת השעון. עם פעולה זו תגרום לעדיפות לרדת מזו של התהליך הבא בתור, תהליך זה יכנס לריצה.

עדיפויות יכולות להינתן באופן סטטי או דינמי. באופן סטטי - במחשב צבאי, תהליך שמופעל על ידי גנרל יתכן ויקבל עדיפות 100, תהליך של קולונל יקבל עדיפות 90, מיגור 80, קפטיין 70 וכן הלאה. מאידך, במרכז מחשבים מסחרי, עבודות בעדיפות גבוהה יעלו 100$ לשעה, עדיפות בינונית תעלה 75$ לשעה ועדיפות נמוכה תעלה 50$ לשעה. למערכת יוניקס יש פקודה, NICE, אשר מאפשרת למשתמש באופן עצמאי להפחית את העדיפות של התהליך שלו, במטרה להיות נחמד nice למשתמשים אחרים. אף אחד לא משתמש בה.

באופן דינמי – עדיפויות יכולות להינתן על ידי המערכת כדי להשיג כמה מטרות. לדוגמא, תהליכים הקשורים מאוד לקלט\פלט ומבלים את רוב זמנם בהמתנה לקלט\פלט שיסתיים. בכל פעם שתהליך שכזה רוצה זמן מעבד, יש לתת לו מעבד באופן מיידי, כדי שיתחיל את הבקשה לקלט\פלט, ואחר כך ניתן להמשיך במקביליות, עם תהליך אחר שממש זקוק לחישובים. המשמעות עבור המעבד של הכרחת תהליכים שקשורים מאוד לקלט\פלט להמתין זמן רב, היא העסקת הזיכרון, זמן רב מהדרוש. באלגוריתם פשוט הנותן שרות טוב לתהליכי קלט\פלט , נקבעת העדיפות על פי נוסחא של 1f/ כאשר f מוגדר כחלק של הזמן האחרון (LAST QUANTUM) שבו התהליך השתמש. תהליך שהשתמש רק ב- 2 מילישניות מתוך ה- 100 שהוקצבו לו יקבל עדיפות 50, כאשר תהליך שפעל 50 מילישניות, לפני שנחסם, נקבל עדיפות 2, ותהליך שהשתמש בכל פרק הזמן שהוקצב לו יקבל עדיפות 1.

לעיתים קרובות נוח לסווג תהליכים לקבוצות לפי עדיפויות ולהשתמש בתזמון לפי עדיפויות בין הקבוצות, אולם בתזמון ROUND ROBIN בתוך הקבוצה. איור 2-25 מציג מערכת עם 4 סווגי קבוצות לפי עדיפויות. אלגוריתם התזמון הוא כזה: כל עוד פועלים תהליכים מקבוצת עדיפות 4, כאשר כל אחד מהם פועל QUANTUM אחד, לפי גישת ROUND ROBIN, ולא מופרע על ידי קבוצות עדיפות נמוכות יותר. אם קבוצת עדיפות 4 ריקה, אזי קבוצת עדיפות 3 תרוץ לפי ROUND ROBIN, אם קבוצה 4 וקבוצה 3 ריקות אזי תרוץ קבוצה 2, וכך הלאה. אם העדיפויות לא ישתנו מזמן לזמן, קבוצת עדיפות נמוכה עלולה לרעוב למוות.

MULTIPLE QUEUES 2.4.3

אחד מאלגורתמים לתזמון המוקדמים ביותר היה CTSS. ל- CTSS הייתה בעיה, שהחלפה בין תהליכים הייתה איטית מאוד, כי ה- 7094 יכל לשמור רק תהליך אחד

בזיכרון. בכל החלפה היה צריך לשמור את התהליך בדיסק, ולקרוא מהדיסק תהליך חדש.

עמוד 66

מתכנני ה-CTSS הבינו שיותר יעיל להקצות לתהליכים הזקוקים להרבה זמן מעבד, QUANTUM גדול אחת לכמה זמן , מאשר להקצות מעט זמן לכל תהליך, לעיתים קרובות (כדי להפחית את ההחלפות). מאידך הקצאת זמן רב לכל התהליכים משמעותו זמן תגובה ארוך, כפי שכבר ראינו. הפתרון שלהם היה לחלק את התהליכים לקבוצות עדיפות. תהליך בקבוצת עדיפות גבוהה, ירוץ במשך קצובת זמן אחת. תהליכים בקבוצת עדיפות יורדת, ירוצו במשך 2 קצובות זמן. תהליכים בקבוצת עדיפות הבאה ירוצו במשך 4 קצובות זמן, וכן הלאה. בכל פעם שתהליך ינצל את כל הזמן שהוקצב לו (כלומר, צורך זמן ריצה גדול) , הוא ירד קבוצה אחת. מתחילים עם תהליך בעדיפות הגבוהה ביותר וקצובת הזמן הקטנה ביותר, וכך בהדרגה אם הוא צורך זמן ריצה גדול הוא ירד בעדיפות, ותועלה לו קצובת הזמן.
לדוגמא, נניח שיש תהליך הזקוק לחישוב בן 100 קצובות זמן. בתחילה הוא יקבל קצובת זמן אחת ויוחלף. בפעם הבאה הוא יקבל 2 קצובות זמן לפני שיוחלף, ואחר כך 1,4,8,16,32- 64 קצובות זמן, למרות שהוא השתמש רק ב- 37 קצובות זמן מתוך ה 64 הסופיים, כדי להשלים את עבודתו. לשם כך נדרשו רק 7 החלפות (כולל הטעינה הראשונית) במקום 100 עם האלגוריתם ROUND ROBIN בלבד. ככל שתהליך יורד בתור העדיפויות, הוא רץ לעיתים רחוקות יותר ויותר, ושומר את זמן המעבד לתהליכים אינטראקטיבים קצרים.
המדיניות הזו אומצה כדי למנוע תהליכים שזקוקים לזמן ריצה רב, כאשר היא החלה, אולם הפכה מאוחר יותר למדיניות עבור תהליכים אינטראקטיבים. בכל פעם שמק RETURN מוקש במסוף, התהליך ששייך למסוף זה עובר לקבוצת העדיפות הגבוהה ביותר, בהנחה שהוא אינטראקטיבי. יום אחד, משתמש כלשהו עם תהליך הדורש זמן רב של מעבד גילה שרק על ידי ישיבה ליד המסוף, והקשת RETURN, באופן אקראי בכל כמה שניות, שיפרה פלאים את זמן התגובה. הוא סיפר לכל חבריו. הלקח בסיפור: לתכנן ולהפעיל את אלגוריתם בפועל קשה הרבה יותר מאשר לתכנן אותו נכון על הנייר.
אלגוריתמים רבים היו בשימוש לסימון תהליכים בקבוצות עדיפות. לדוגמא, ל-XDS 940 היו 4 קבוצות עדיפות שנקראו, מסוף, קלט\פלט, קצובות זמן קצרות, וקצובות זמן ארוכות. כאשר תהליך שחיכה לקלט ממסוף הוער לבסוף הוא קיבל את העדיפות הגבוהה ביותר (מסוף). כאשר תהליך הממתין לבלוק בדיסק היה מוכן, הוא קיבל את העדיפות השניה. כאשר תהליך עדיין היה בריצה כשהסתיימה קצובת זמן, הוא קיבל עדיפות שלישית. אולם, אם תהליך ניצל את קצובת הזמן שלו מספר רב מידי של פעמים ברצף, מבלי להיחסם עבור מסוף או קלט\פלט, הוא הועבר למטה לתחתית התור. מערכות רבות השתמשו במשהו דומה כדי להעדיף משתמשים אינטראקטיביים.

עמוד 67

SHORTEST JOB FIRST 2.4.4

רוב האלגוריתמים שפורטו תוכננו עבור מערכות אינטראקטיביות. כעת בוא נתבונן באלגוריתם שמתאים במיוחד לעבודות אצווה (BATCH), כאשר זמני הריצה ידועים מראש. בחברת ביטוח, לדוגמא, ניתן לחזות כמעט במדויק כמה זמן ייקח זמן ריצה של 1000 תביעות, היות ועבודה דומה כזאת נעשית בכל יום. כאשר כמה עבודות בעלות אותה חשיבות נמצאות בתור הקלט, ממתינות לאתחול, המתזמן צריך להשתמש ב- SHORTEST JOB FIRST . התבונן באיור 26-2. כאן מצאנו 4 עבודות A,B,C,D בעלי זמן ריצה 8,4,4,4 דקות, בהתאמה. על ידי הרצתם בסדר זה, A יתבצע תוך 8 דקות, B יתבצע תוך 12 דקות, C יתבצע תוך 16 דקות ו- D יתבצע תוך 20 דקות. זמן ממוצע 14 דקות.
כעת ננסה להריץ עבודות אלו מהקצר לארוך. זמן הביצוע יהיה עכשיו 4,8,12,20 דקות, ובממוצע 11 דקות. ביצוע העבודות הקצרות קודם הוכח כאופטימלי. כלומר כאשר אנו מסדרים את התהליכים מזמן ריצה הקצר לארוך מתקבל זמן ביצוע ממוצע מינימלי. מסיבה זו, היה נחמד לו יכולנו להשתמש באלגוריתם עבור תהליכים אינטראקטיבים גם כן. בהיקף מסוים ניתן. לתהליכים אינטראקטיבים ישנו דפוס מסוים של המתנה

לפקודה, ביצוע פקודה, המתנה לפקודה, ביצוע פקודה, וכן הלאה. אם נתייחס לכל פקודה כ״עבודה נפרדת, אזי נוכל למזער את זמן התגובה הכללי. הבעיה היחידה היא לחשב איזה תהליך מבין התהליכים הפועלים הוא הקצר ביותר.

גישה אחת היא לבצע הערכה המבוססת על התנהגות קודמת, ולהריץ את התהליך עם ההערכה הנמוכה ביותר לזמן ריצה. נניח שהזמן שהוערך לפקודה, עבור מסוף מסוים הוא T0. כעת נניח שבריצה הבאה הוא יוערך ב- T1. אנו נוכל לעדכן את ההערכה לפי הנוסחא aTO+(1-a)T1 . דרך הברירה a אנו יכולים להחליט האם לזכור את ההערכות הקודמות זמן רב, או לשכחן מהר.

עמוד 68

עם a=1\2 נקבל את ההערכות: T0, T0\2+T1\2, T0\4+T1\4+T2\2, T0\8+T1\8+T2\4+T3\2

לפיכך, לאחר 3 ריצות חדשות, המשקל של T0 בשערוך החדש יורד ל- 1\8.

שיטת שערוך הערך הבא הוא סדרה, הנקבעת על ידי לקיחת המשקל הממוצע של ערך השערוך הנוכחי עם הערכים הקודמים, והיא נקראת AGING. והוא מתאים לסיטואציות רבות כאשר יש לבצע צפי על סמך ערכים קודמים. AGING קל לביצוע במיוחד כאשר a=1\2. כל מה שצריך הוא להוסיף את הערך החדש לשערוך הנוכחי, ולחלק את הסכום ב- 2 (על ידי הזזת הביט הימני).

יש לציין כי האלגוריתם הראשון של ביצוע עבודות מהזמן הקצר, לארוך הוא אופטימלי רק כאשר כל העבודות פנויות בו זמנית. דוגמא נגדית, נניח 5 עבודות מ- A ועד E , עם זמן ביצוע של 2, 4, 1, 1 בהתאמה. זמן ההגעה שלהם הוא 0, 0, 3, 3, 3. בתחילה רק A או B יכולים להיבחר, מאחר ושלושת העבודות האחרות עדיין לא הגיעו. שימוש באלגוריתם SHORTEST JOBS יריץ את העבודות בסדר הבא: A,B,C,D,E בהמתנה ממוצעת של 4.6. אולם, לביצוע הפעולות בסדר B,C,D,E,A זמן המתנה של 4.4.

2.4.5 תזמון מובטח

גישה אחרת לגמרי לתזמון היא לבצע מראש פשרה בין המשתמשים, תוך הבטחה מראש לגבי זמן ביצוע. אם יש N משתמשים שנכנסו למערכת, נותנים לכל אחד מהם 1\N מהמעבד.

כדי לקיים את ההבטחה, על המערכת לעקוב כמה זמן מעבד ניצל כל משתמש, עבור כל התהליכים שלו מאז שנכנס למערכת, וכן כמה זמן כל משתמש נמצא במערכת. אזי הוא מחשב את זמן המעבד שלו זכאי כל משתמש, פשוט על ידי חילוק הזמן מאז הכניסה ב- N. מאחר וידוע זמן ניצול המעבד, של כל משתמש, ניתן לחשב את הפרופורציה בין זמן מעבד לו זכאי המשתמש, לבין הזמן שהוא ניצל בפועל. משמעות של יחס 0.5 היא שתהליך ניצל רק חצי ממה שיכל לנצל. משמעות יחס 2.0 היא שתהליך ניצל כפול מהזמן המוקצב לו. האלגוריתם מריץ את התהליך עם היחס הנמוך ביותר עד אשר היחס שלו עולה מעל התהליך המתחרה לו.

המטרה היא למנוע הרעבה, בכל מקרה – שיקול ההגינות עומד בשיטה זו מעל הכל. ניתן ליישם רעיון דומה במערכות זמן אמת, אשר לתהליכים יש מועד סופי לתום הביצוע. כאן אנו מחפשים את התהליך עם הסכנה הגדולה ביותר, להחמצת המועד הסופי, ומריצים אותו קודם. תהליך שחייב לסיים בתוך 10 שניות, מקבל עדיפות על כזה שחייב לסיים בתוך 10 דקות.

2.4.6 POLICY VERSUS MECHANISM

עד עתה, הנחנו הנחה סמויה שכל התהליכים במערכת שייכים למשתמשים שונים, ולכן מתחרים על זמן מעבד. למרות שבדרך כלל הדבר נכון, לפעמים קורה שתהליך מסוים ובניו רצים תחת שליטתנו.

עמוד 69

לדוגמא, במערכת לניהול מאגר נתונים, לתהליך אפשר שיהיו הרבה בנים. כל ילד יטפל בבקשה נפרדת, או שלכל אחד מהם יהיה פונקציה נפרדת לבצע. בהחלט יתכן שלתהליך הראשי יש דעה ברורה, מי מהבנים חשוב יותר (שזמן הביצוע שלו קריטי) ומי פחות. לדאבוננו, אף אחד מהאלגוריתמים לתזמון שדנו בהם לעיל, לא מקבל קלט מהמשתמש

לגבי החלטות תזמון. כתוצאה מכך, המתזמן, באופן נדיר מבצע את הברירה הטובה ביותר.
הפתרון לבעיה זו הוא להפריד את מכניזם התזמון ממדיניות התזמון. כשהכוונה היא שהאלגוריתם לתזמון בנוי בצורת פרמטרים, אולם את הפרמטרים הללו ניתן למלא בתהליכי משתמש. למשל נשתמש בדוגמת מאגר הנתונים שוב. נניח שהגרעין משתמש באלגוריתם עדיפויות לתזמון, אולם מספק למשתמש קריאות מערכת שבעזרתם הוא יכול לשנות את דרגות העדיפות של הבנים. בדרך זו יכול הורה לשלוט לפרטי פרטים כיצד לתזמן את הבנים, אפילו שהוא עצמו לא מבצע כל תזמון. כאן המכניזם הוא בגרעין, אולם המדיניות ניתנת לתהליך המשתמש.

2.4.7 TWO-LEVEL SCHEDULING

עד עתה פחות או יותר הנחנו שכל התהליכים הניתנים לריצה שמורים בזיכרון הראשי. אם הזיכרון אינו מספיק, נצטרך לשמור חלק מהתהליכים בדיסק. לעובדה זו יש השפעה רבה על תזמון תהליכים מאחר שהזמן שלוקח להביא תהליך מהדיסק לריצה גדול יותר, מאשר החלפה בין תהליכים שכבר נמצאים בזיכרון. דרך מעשית יותר להתמודד עם ההחלפה היא להשתמש בתזמון בשתי רמות. חלק מהתהליכים יטענו לתוך הזיכרון ואז המתזמן יבצע תזמון תהליכים רק ברמה זו. לאחר פרק זמן מסוים מתזמן ברמה גבוהה יותר יזיז תהליכים שכבר שהו זמן מספיק בזיכרון ויטען תהליכים ששהו על הדיסק זמן מספיק. כאשר שנוי כזה מבוצע (ראה איור 2-27) המתזמן ברמה הנמוכה, מגביל את עצמו להרצת תהליכים שנמצאים בזיכרון בלבד. לפיכך, המתזמן ברמה הנמוכה אחראי ביצוע החלטות לגבי תהליכים שנמצאים בזיכרון כרגע. בעוד המתזמן ברמה הגבוהה מופקד על העברת תהליכים אל ומהדיסק לזיכרון.
הקריטריונים לפיהם יכול להשתמש המתזמן ברמה הגבוהה, על מנת לקבל החלטות הם:

1. כמה זמן עבר מאז שתהליך עבר החלפה, פנימה והחוצה?
2. כמה זמן מעבד צרך התהליך לאחרונה.
3. כמה גדול התהליך? (תהליכים קטנים לא מפריעים).
4. מה עדיפות התהליך.

ושוב, כאן אנו יכולים להשתמש ב-ROUND ROBIN, תזמון עדיפויות, או כל שיטת תזמון אחרת.

עמוד 74

פרק 3 : ניהול זיכרון

זיכרון הוא משאב חשוב שיש לנהלו בזהירות. בעוד למחשב הביתי הממוצע כיום יש פי עשר יותר זיכרון מאשר ל-IBM 7094 המחשב הגדול ביותר בעולם, בתחילת שנות השישים, תוכניות מקבלות זיכרון גדול יותר ומהיר יותר. בפרק זה נלמד כיצד מערכת ההפעלה מנהלת את הזיכרון.
החלק במערכת ההפעלה המופקד על ניהול הזיכרון נקרא MEMORY MANAGER.
תפקידו לעקוב אחר איזה חלקים בזיכרון בשימוש, ואיזה לא, להקצות לתהליכים זיכרון כשהם זקוקים לו , ולקחת אותו כשאינם זקוקים לו, לבצע החלפות בין תהליכים בזיכרון הראשי לדיסק כאשר הזיכרון הראשי לא גדול מספיק.
בפרק זה נחקור מספר אופני ניהול זיכרון, החל מהפשוט ביותר עד למתוחכם. נתחיל במערכת ניהול הזיכרון הפשוטה ביותר האפשרית, ואז בהדרגתיות נתקדם ליותר משוכללים.

3.1 ניהול זיכרון ללא החלפה או דפדוף

מערכות לניהול זיכרון נחלקות לשתי מחלקות עיקריות :

1. אלו שלא מעבירות תהליכים הלוך וחזור בין הזיכרון הראשי, לבין הדיסק במשך הביצוע (החלפה ודפדוף).
2. אלו שמבצעות החלפה ודפדוף.

במהלך פרק זה על הקורא לזכור שהחלפה ודפדוף הם תוצר לוואי הנגרם מחוסר בזיכרון ראשי שיכול להכיל את כל התוכניות יחד. ביום שבו תיפתר בעיה זו, לא יהיה עוד צורך בהפעלתן, ונעדיף ניהול במקום אחד.

עמוד 75

3.1.1 תוכנית יחידה (ללא החלפה או דפדוף)

הדרך הפשוטה ביותר לניהול זיכרון היא להחזיק רק תהליך אחד בכל זמן נתון, ולהתיר לתהליך להשתמש בכל הזיכרון. המשתמש טוען את תוכנית מהדיסק, או מטייפ, והיא משתלטת על כל המחשב. אמנם גישה זו הייתה מקובלת בתחילת 1960, היא אינה בשימוש כיום, אפילו לא במחשבים ביתיים זולים, בעיקר היות והיא דורשת שכל תהליך יכיל בתוכו כונן לכל התקן קלט\פלט שבשימוש.
השימוש בטכניקה הרגילה על מיקרו מחשבים פשוטים מודגם באיור 3-1. הזיכרון מחולק בין מערכת ההפעלה, ותהליך משתמש יחיד.
לפי גישה זו תיתכנה שלוש דרכים לניהול הזכרון בין תוכנית יחידה ואחסון מערכת ההפעלה :

1. מערכת ההפעלה נמצאת ב-RAM (קריאה וכתיבה) ותוכנית המשתמש מעליה.
2. מערכת ההפעלה נמצאת ב-ROM (קריאה בלבד) ותוכנית המשתמש מעליה.
3. מערכת ההפעלה ברובה נמצאת ב-RAM, מעליה תוכנית המשתמש ומעליה מאוחסנים התקני מערכת ההפעלה ב-ROM. זו הגישה המקובלת בדוס.

כאשר המערכת מאורגנת בצורה שכזו, רק תהליך יחיד יכול לרוץ בזמן נתון. המשתמשים מקישים פקודה במסוף, ומערכת ההפעלה טוענת את התוכנית המבוקשת מהדיסק לזיכרון ומבצעת אותה. עם סיום ביצוע תהליך, מערכת ההפעלה מדפיסה תו עזר כלשהו על המסך וממתינה לפקודה מהמסוף לטעון תהליך נוסף, תוך מחיקת הראשון.

עמוד 76

3.1.2 ריבוי תוכניות וניצול הזיכרון

אף על פי שתוכנית יחיד נמצאת לעיתים בשימוש במחשבים קטנים, במחשבים גדולים עם הרבה משתמשים היא נדירה. סיבות לשימוש בריבוי תוכניות :

1. כדי להקל על תכנות יישום על ידי חלוקת התוכנית לכמה תהליכים.
2. מחשבים גדולים לעיתים קרובות נותנים שרות לכמה אנשים במקביל, דבר זה מביא לדרישה שיותר מתהליך אחד ישהה בזיכרון בפרק זמן נתון. טעינת תהליך

מהדיסק אינה יעילה כאשר התהליך רץ זמן מועט – 100 מילי-שניות. אולם אם קצובת הזמן לתהליך נקבעת מעל 100 מילי-שניות אזי זמן התגובה ארוך מאוד.

3. רוב התהליכים מבלים חלק ניכר מזמנם כאשר הם ממתינים לאמצעי קלט\פלט. בשיטה זו של ריבוי תוכניות המעבד ינוצל טוב יותר. (בשיטה של תוכנית אחת המעבד ימתין 80% מזמנו).

<u>המודל של ריבוי תוכניות:</u>

<u>המודל הבסיסי</u>: כאשר משתמשים בריבוי תוכניות, ניצול המעבד משתפר. שכן אם תהליך ממוצע מחשב (צורך זמן מעבד) רק 20% מהזמן בו הוא נמצא בזיכרון (בשאר הזמן הוא ממתין לאמצעי קלט\פלט), עם 5 תהליכים בבת אחת בזיכרון, המעבד ינוצל במלואו. מודל זה אופטימי בצורה לא ריאלית, שכן הוא מניח שהתהליכים אף פעם לא ממתינים יחדיו להתקן קלט\פלט.

<u>מודל משופר יותר</u> מסתכל על המעבד מנקודת מבט הסתברותית. נניח שתהליך מנצל חלק P מהזמן בהמתנה לאמצעי קלט\פלט. עם N תהליכים בזיכרון בבת אחת, ההסתברות שכל N התהליכים ממתינים לאמצעי קלט\פלט הוא Pבחזקת N. => ניצול זמן המעבד הוא PN- 1 (1 פחות P בחזקת N). איור 2-3 מראה את ניצול המעבד כפונקציה של N, אשר נקרא הדרגה של ריבוי התוכניות.

לפי מודל זה ברור שאם תהליך ממתין 80% מזמנו לאמצעי קלט\פלט, לפחות 10 תהליכים צריכים לשהות בזיכרון ביחד כדי שבזבוז זמן המעבד יקטן מ- 10% . כאשר מבינים שתהליך אינטראקטיבי הממתין למשתמש שיקליד משהו, זהו גם זמן המתנה לאמצעי קלט\פלט, אזי ברור שזמן המתנה לקלט\פלט של 80% ויותר, הוא שיגרתי ביותר. אולם אפילו במערכות אצווה, תהליכים מבצעים הרבה קלט\פלט בדיסק או בטייפ, ימתינו לעיתים קרובות אחוז זה מהזמן ויותר.

עמוד 77 שרטוט 2-3 מציג את הדיאגרמה של ריבוי התוכניות וניצול המעבד.

לפי מודל זה ככל שמספר התהליכים גדול יותר, ניצול המעבד טוב יותר (עד מספר תהליכים מסוים ששם הגרף נהיה אופקי).

לפי הדיאגרמה ניתן למצוא באחוז מסוים של המתנה לאמצעי קלט\פלט – מה מספר התהליכים שצריכים להיות בזיכרון על מנת להגיע לניצול מקסימלי של המעבד.

וביתר דיוק, המודל ההסתברותי עוסק בהערכות וקירובים. המודל מניח שכל התהליכים הם עצמאיים כך שייתכן שכמה תהליכים רצים בו זמנית, דבר שלא אפשרי במנגנון שבו שיש מעבד אחד (התהליכים יאלצו להמתין למרות שהם מוכנים לריצה). למשל במערכת עם 5 תהליכים די הגיוני ש- 3 תהליכים ירוצו ו- 2 ימתינו, אולם במחשב עם מעבד אחד רק תהליך אחד יכול לרוץ.

<u>עמוד 77</u>

לפיכך התהליכים אינם עצמאיים. ניתן לתכנן מודל מדויק יותר המשתמש בתורת התורים. אולם ניתן להשתמש במודל 2-3 לניבוי ביצועי המעבד (הערכות וקירובים).

נניח למשל שלמעבד יש 1 מגה זיכרון, עם מערכת הפעלה הצורכת 200K . הגדלים הללו מאפשרים ל- 4 תהליכים לשהות בזיכרון בבת אחת. עם ממוצע המתנה לאמצעי קלט\פלט של 80%, נקבל ניצול מעבד של 60% בקירוב. תוספת של עוד 1 מגה לזיכרון, תאפשר למערכת להריץ 9 תהליכים יחד, מה שיעלה את ניצול המעבד ל- 87%. עוד תוספת של 1 מגה יעלה את ניצול המעבד ל- 96%, כלומר התפוקה תעלה ב- 10 אחוזים בלבד. על ידי השימוש במודל זה, בעל המחשב יוכל להחליט שהתוספת השניה לזיכרון כדאית ואולם התוספת השלישית אולי לא.

<u>ניתוח ביצועי מערכת רבת-תוכניות</u>

מודל זה יכול להיות שימושי בניתוח מערכות אצווה (BATCH). נניח לדוגמא, מרכז מחשבים שממוצע המתנה שלו לאמצעי קלט\פלט הוא 80%. בבוקר מסוים נמסרו 4 עבודות (ראה איור 3-3. העבודה הראשונה שהגיעה ב- 10 בבוקר דורשת 4 דקות של זמן מעבד. עם זמן המתנה לאמצעי קלט\פלט של 80% העבודה מנצלת רק 12 שניות מזמן המעבד, לכל דקה בה הוא נימצא בזיכרון, אפילו אם כל עבודה אחרת לא מתחרה אתו על זמן מעבד. שאר 48 השניות מבוזבזות על המתנה לקלט\פלט שיסתיים. לפיכך העבודה תצטרך לשהות בזיכרון לפחות 20 דקות על מנת לקבל 4 דקות של עבודת מעבד, אפילו בהעדר תחרות על המעבד.

מהשעה 10 עד 10:10 העבודה הראשונה שוהה לבד בזיכרון ו- 2 דקות של עבודה מבוצעות.

עמוד 78

כאשר העבודה השניה מגיעה בשעה 10:10, ניצול המעבד עולה מ- 20% ל- 36% בזכות ריבוי התוכניות. אולם, עם תזמון ROUND ROBIN, כל עבודה מקבל חצי מהמעבד, כך שכל עבודה יש 0.18 דקות של עבודה מבוצעת לכל דקה בזיכרון. שימו לב שהתוספת של העבודה השניה עלתה לעבודה הראשונה רק ב- 10% מביצועיו (מ- 0.20 ל- 0.18 דקות של מעבד לכל דקה בפועל). עם תוכנית שלישית כל עבודה תקבל 0.16 דקות מעבד לכל דקה בפועל, כפי שמוצג באיור 3-3. מהשעה 10:15 ועד 10:20 כל אחת משלוש התוכניות מקבלת 0.8 דקות של זמן מעבד. בשעה 10:20 התוכנית הרביעית מצטרפת. רצף האירועים מוצג באיור 3-3.

3.1.3 חלוקה למחיצות קבועות

עתה, צריך כבר להיות ברור שלעיתים קרובות יותר יעיל, להחזיק יותר מתהליך אחד בזיכרון בזמן נתון. השאלה היא אם כך: "איך נארגן את הזכרון על מנת להשיג מטרה זו?" הדרך הפשוטה ביותר היא לחלק את הזיכרון ל- N חלקים (נניח על ידי מערכת ההפעלה בעת אתחול המחשב).

החלקים בדרך כלל הם בעלי גודל שונה זה מזה וכאשר תהליך מגיע הוא מצטרף לתור של תהליכים ממתינים, לבלוק הקטן ביותר שיכול להכיל אותו.

עמוד 79

היות והחלקים קבועים מראש, אם נותר רווח בבלוק זה שלא נוצל על ידי התהליך שהוכנס הוא הולך לאיבוד. איור 3-4 מראה כיצד שיטה זו של חלקים קבועים, ותור נפרד של קלט, פועלת. לפי מבנה זה לכל בלוק יש תור נפרד משלו.

החיסרון בחלוקת התוכניות הנכנסות לתורים נפרדים נעשה ברור כאשר תור לחלק שגודלו גדול בזיכרון, ריק, אולם התור לבלוק קטן בזיכרון, מלא.

מבנה אלטרנטיבי הוא בעל תור אחד לכל הבלוקים. כאשר בלוק מתפנה, התוכנית הקרובה ביותר לראש התור שיכולה להיכנס לבלוק, יכולה להיטען לבלוק הפנוי, ולרוץ.

החיסרון במבנה זה הוא בזבוז בלוקים גדולים על תהליכים קטנים.

אסטרטגיה אחרת שמתגברת על החסרון הנ"ל היא, שכאשר מתפנה בלוק, לחפש בכל תור הקלט. התהליך הגדול ביותר שממתין, יכנס. החיסרון במבנה זה הוא אפליה לרעה של תוכניות הצורכות פחות זיכרון, כי לא כדאי לבזבז עליהם בלוק שלם. כאשר בדרך כלל עדיף לתת לתוכניות קטנות (בהנחה שהן אינטראקטיביות) את השרות הטוב ביותר, ולא הגרוע ביותר.

דרך אחת להתגבר על חיסרון זה היא להחזיק לפחות בלוק אחד קטן שיאפשר לתהליכים קטנים לרוץ בלי צורך להקצות להם בלוקים גדולים.

גישה נוספת, היא להחליט על חוק שאומר שלא ניתן לדלג על תהליך מעבר ל- K פעמים. בכל פעם שמדלגים עליו הוא מקבל נקודה אחת. כאשר הוא מגיע ל- K נקודות לא ניתן לדלג עליו.

מערכת זו, עם בלוקים קבועים, הנקבעים על ידי המפעיל, קלה להבנה ולביצוע. התהליכים נכנסים לתור עד שבלוק מתאים מתפנה, בכל פעם תהליך אחד נטען לבלוק ומורץ עד לסיומו.

עמוד 80

הזזה והגנה – RELOCATION AND PROTECTION

ריבוי תוכניות מעלה שתי בעיות מרכזיות שיש לפותרן – הזזה והגנה. ראה איור 3-4. ממנו ניתן להבין שתהליכים שונים רצים בכתובות שונות. כאשר התוכנית עוברת קומפילציה ומגיעה לשלב הקישור (LINK), ה-LINKER חייב לדעת מאיזה כתובת בזיכרון תתחיל התוכנית.

לדוגמא, נניח שההוראה הראשונה היא קריאה לשגרה בכתובת 100 עם קובץ בינארי שנוצר על ידי ה-LINKER. אם תוכנית זו נטענת בבלוק 1, הוראה זו תקפוץ לכתובת אבסולוטית 100, שנמצאת בתוך מערכת ההפעלה. לכן החישוב הוא 100K+. אם התוכנית נטענת לבלוק 2 אזי יש לבצע 100 + 200K וכן הלאה. בעיית חישוב הכתובת הפיזית בהתאם למיקום בבלוק קרויה הזזה.

פתרון אחד הוא ממש לשנות את ההוראות כאשר התוכנית נטענת לזיכרון. לתוכניות הנטענות לבלוק 1 יש להוסיף 100K לכל כתובת. תוכניות הנטענות לבלוק 2 יוסיפו 200K

לכל כתובת. לשם כך ה-LINKER חייב להכיל בתוכנית הבינארית שלו רשימה או מפה של ביטים המכיל אינפורמציה לגבי איזה תוכניות צריכות לעבור הזז ואיזה לא. למעשה משתמשים בכתובות בקוד שהן יחסיות לנקודת ההתחלה ולא בכתובות אבסולוטיות – על ידי כך הקוד נייד ומאפשר הזזה.
הזזה במשך הטעינה לא פותרת את בעיית ההגנה. תוכנית זדונית תוכל תמיד לתכנן הוראות חדשות כדי להגיע למיקום מסוים בזיכרון. כי תוכניות במערכת הזו משתמשות בזיכרון אבסולוטי במקום בזיכרון יחסי לאוגר, אין דרך לעצור תוכנית מלבנות הוראות הקוראות או כותבות כל מילה בזיכרון. במערכת רבת משתמשים, אסור לאפשר לתהליכים לקרוא או לכתוב בזיכרון השייך למשתמשים אחרים.
הפתרון של IBM הוא לחלק את הזיכרון לבלוקים של 2KB והצמדה של קוד בין 4 סיביות לכל בלוק שיהווה עבורו הגנה. היות ורק מערכת ההפעלה יכולה לשנות את קוד ההגנה, נמנע מתהליך משתמש להתערב בתהליך אחר וכן גם במערכת ההפעלה.
פתרון נוסף לבעיית ההזזה וההגנה גם יחד הוא לצייד את החומרה בשני אוגרים מיוחדים הנקראים BASE ו- LIMIT. כאשר תהליך נטען אוגר BASE נטען עם כתובת ההתחלה של הבלוק בזיכרון, ואוגר LIMIT טוען את אורך הבלוק.
בדרך זו ניתן לחשב את הכתובת הפיזית: כתובת יחסית של התוכנית+BASE וכן מתבצעת בדיקה של האוגר LIMIT בכדי לבדוק שתוכנית לא גלשה מתחום הזיכרון שלה (עבור הגנה). למשל אם אוגר BASE הוא 100K, קריאה לכתובת 100 היא למעשה קריאה לכתובת 100 +100K , ללא שינוי הכתובת בפועל. החומרה מגינה על אוגרים אלו כדי למנוע מתוכניות משתמש לשנות אותם.

עמוד 81

יתרון נוסף בשימוש באוגר BASE הוא שניתן להזיז את התוכנית במרחב הזיכרון לאחר שהיא החלה לרוץ. כל שדרוש הוא לשנות את אוגר BASE. כאשר ההזזה מבוצעת על ידי שינוי הכתובות עם טעינת התוכנית, לא ניתן להזיזה מבלי לעבור על כל תהליך השינוי מחדש.
*הקוד היחסי הוא תנאי הכרחי למימוש המכונה המדומה. קוד מוחלט הוא תנאי מכונה, ואינו נייד.

3.2 החלפה בזיכרון – SWAPPING

במערכות אצווה, ארגון הזיכרון בבלוקים קבועים הוא פשוט ויעיל. כל עוד ישנן תוכניות בזיכרון המעסיקות את המעבד כל הזמן, אין צורך להשתמש במשהו מסובך יותר.
בחלוקה לפי זמן (TIMESHARING) המצב הוא שונה: יש יותר משתמשים מאשר יש זיכרון לשמור את התהליכים שלהם, כך שהכרחי לשמור את התהליכים על הדיסק. כדי להריץ את התהליכים הללו יש להביאם לזיכרון. העברת תהליכים מהזיכרון הראשי לדיסק וחזרה נקרא SWAPPING.

3.2.1 ריבוי תוכניות עם בלוקים ניידים.

בעקרון, מנגנון ההחלפה יכול להתבסס על בלוקים קבועים בזיכרון. כאשר תהליך נחסם, ניתן להזיזו לדיסק ותהליך אחר יובא לבלוק מהדיסק. בפועל בלוקים קבועים אינם יעילים כאשר הזיכרון הוא קטן היות ויותר מידי ממנו מבוזבז על ידי תוכניות קטנות מהבלוקים שהן מאוחסנות בהם. לכן משתמשים באלגוריתם שונה לניהול זיכרון. והוא נקרא VARIABLE PARTITIONS. בלוקים ניידים, שמאפשרים לנצל את הבלוקים עד תום.
לפי שיטה זו מספר הבלוקים וגודלם בזיכרון משתנה במשך היום. איור 3-5 מראה כיצד פועלת השיטה. בתחילה רק תהליך A נמצא בזיכרון. אחר כך תהליכים B ו- C נוצרים או מועברים מהדיסק. A מופסק ומועבר לדיסק. אחר D מגיע ו- B יוצא ולבסוף E נכנס.
ההבדל העיקרי בין הבלוקים הקבועים באיור 3-4 לבין הבלוקים המשתנים באיור 3-5 הוא שמספרם, מיקומם וגודלם של הבלוקים משתנה, כאשר תהליכים באים והולכים, בעוד שבאיור הקודם הם קבועים.
הניידות במספר הבלוקים שיכול להיות גדול מדי או קטן מדי משפר את ניצול הזיכרון אולם מסבך את נושא הקצאת הכתובות בזיכרון, ושמירת המעקב אחריהן.

יש אפשרות לסגור פערים – רווחים בבלוקים על ידי הזזה של כל התהליכים כלפי מטה, ככל שהדבר ניתן. שיטה זו נקראת MEMORY COMPACTION, - דחיסת הזיכרון.
בדרך כלל לא מבצעים זאת מכיוון שזה דורש הרבה זמן מעבד. לדוגמא במחשב עם 1MB שיכול להעתיק בית אחד למיקרו-שניה, לוקח דקה אחת לדחוס את כל הזיכרון. כלומר בדרך כלל לא מבצעים זאת מכיוון שזה דורש הרבה זמן מעבד.
כאן עולה השאלה כמה זיכרון יש להקצות לתהליך כאשר הוא נוצר או מועבר מהדיסק. אם תהליך נוצר בגודל שלעולם לא משתנה ההקצאה היא פשוטה, מקצים בדיוק כמה שצריך, לא יותר, לא פחות.

<u>עמוד 82</u>

אולם אם DATA יכולה לגדול, למשל על ידי הקצאה דינמית של זיכרון מתוך ערימה, כפי שניתן בשפות תכנות רבות, מתעוררת בעיה בכל פעם שתהליך מבקש לגדול. אם יש מרחב זיכרון קרוב לתהליך, ניתן להקצות ממנו לתהליך, והוא יוכל לגדול לתוכו. מאידך, אם תהליך קרוב לתהליך שני, גדילה תגרום או שהתהליך הגדל יועבר למרחב זיכרון גדול מספיק עבורו, או שאחד או יותר מהתהליכים יועבר לדיסק, על מנת לפנות מקום לתהליך שגדל. אם תהליך לא יכול לגדול בזיכרון, והדיסק מלא, התהליך יאלץ להמתין, או להפסיק את פעולתו עקב מחסור בזיכרון.
אם ניתן לצפות שרוב התהליכים יגדלו בזמן ריצתם, עדיף להקצות להם יותר מקום, בזיכרון, כדי להפחית את הפעולות שקשורות בהזזת או החלפת תהליכים, שהמקום בזיכרון כבר צר עבורם. אולם, כאשר מעבירים תהליך לאחסון בדיסק יש להקצות מקום בדיוק לפי גודל התהליך. באיור 3-6 - הקצאה אם מרווחים.
אם לתהליך יכולים להיות שני חלקים שגדלים, לדוגמא, חלק ה- DATA וחלק המחסנית, יעיל יותר להשאיר להם חלק אחד משותף לגדילה, כך שה-DATA יגדל כלפי מעלה והמחסנית תגדל כלפי מטה. שרטוט 3-6. במידה ואוזל הזיכרון בין שניהם, כל אחד מהם יכול או לעבור למרחב אחר של זיכרון גדול מספיק, או ללכת לדיסק בהמתנה שיתפנה גודל זיכרון מתאים, או להפסיק את פעולתו מחוסר זיכרון.
באופן כללי למערכת ההפעלה יש שלוש דרכים לעקוב אחרי השימוש בזיכרון: מפת סיביות BIT MAPS, רשימות, שיטת הזוגות BUDDY SYSTEMS.

<u>עמוד 83</u>

<u>ייצוג הזיכרון עם מפת סיביות</u>

עם מפת סיביות, הזיכרון מחולק ליחידות ההקצאה שגודלם יכול לנוע בין כמה מילים לבין כמה KILOBYTS. לכל יחידת הקצאה מוקצה סיבית במפת הסיביות, כאשר היא 0 היחידה פנויה, כאשר היא 1 היחידה תפוסה (או להפך). איור 3-7 מראה את הקשר בין מפת סיביות לבין הבלוקים בזיכרון.
הגודל של יחידות ההקצאה הוא נושא חשוב לתכנון. ככל שהיחידות קטנות יותר, המפה גדלה. אולם, אפילו אם יחידות הקצאה קטנות למשל בגודל 4 בתים, שהם 32 סיביות של זיכרון, הגודל במפה יהיה סיבית אחת בלבד. זיכרון של 32N סיביות ידרוש N סיביות במפה.

<u>עמוד 84</u>

כלומר, סיבית במפה היא החלק 1\33 מהזיכרון. אם יחידת ההקצאה גדולה יותר, חלקה של הסיבית במפה יהיה קטן יותר, אולם יחידות גדולות עלולות להוות בזבוז אם גודל התהליך לא מתאים.
מפת סיביות היא דרך פשוטה לעקוב אחר ניהול הזיכרון בהקצאות קבועות בגלל שגודל המפה תלוי בגודל הזיכרון וגודל יחידת ההקצאה.
החיסרון המרכזי של שיטה זו הוא במעבר על כל מפת הסיביות לחיפוש K ביטים רצופים שווים ל- 0 על מנת לאחסן תהליך בגודל K. לכן שיטה זו אינה שימושית (איטית וגודלה של המפה עלול להיות ניכר למדי).

<u>3.2.3 ניהול זיכרון באמצעות רשימות מקושרות</u>

ייצוג הזיכרון על ידי רשימה מקושרת של סגמנטים מוקצים בזיכרון, וזיכרון פנוי. עמוד 83 שרטוט 3-7. כלומר האם הסגמנט תפוס בתהליך, או שהוא חור בין שני תהליכים.
הזיכרון המיוצג באיור 3-7(a) מיוצג באיור 3-7(c) כרשימה מקושרת של סגמנטים. כל

כניסה ברשימה מייצגת חור או תהליך, הכתובת (הפיזית) של התחלת הסגמנט, גודל הסגמנט ומצביע לכניסה הבאה.
בדוגמא, רשימת הסגמנטים מסודרת לפי סדר הכתובות הפיזיים בזיכרון (מתחילת הזיכרון ועד סופו).
בדוגמא שלנו אחסון רשימת הסגמנטים ממוין לפי כתובות. היתרון שבמיון הוא שכאשר תהליך מסיים או מוחלף, עדכון הרשימה הוא מההתחלה לסוף. לתהליך שהסתיים יש בדרך כלל 2 שכנים (למעט כאשר הוא נמצא ממש בתחתית או תקרת הזיכרון). יש 4 אפשרויות למיזוג בין תהליכים וחורים המודגמים באיור 8-3. עדכון הזיכרון יהיה יעיל יותר אם הרשימה תהיה דו-כיוונית. מבנה זה יקל למצוא כניסה קודמת, כדי לבדוק אם מיזוג יחידות הוא אפשרי.
כאשר תהליכים וחורים נשמרים ברשימה ממוינת לפי כתובות, ניתן להשתמש במספר אלגוריתמים להקצות זיכרון לתהליך חדש (שנוצר עתה, או שהובא מהדיסק). כאשר אנו מניחים שמנהל הזיכרון יודע כמה זיכרון יש להקצות.

<u>אלגוריתמים להכנסת תהליך לרשימה:</u>

<u>FIRSE FIT</u> - מנהל הזיכרון מבצע סריקה מתחילת הרשימה עד שהוא מוצא סגמנט גדול מספיק להכיל את התהליך. הסגמנט הפנוי שנמצא נחלק לשניים – חלק אחד עבור אחסון התהליך, והחלק הנותר נשאר פנוי (בגלל שלרוב גודל התהליך אינו מתאים בדיוק לסגמנט ונותר מקום פנוי). אלגוריתם זה הוא מהיר היות והוא סורק מעט ככל האפשר.

עמוד 85

<u>NEXT FIT</u> – פועל בדומה לקודם. ההבדל הוא שהוא שומר את המקום האחרון בו הוא מצא מקום פנוי בזיכרון. בפעם הבאה הוא מתחיל את הסריקה מהמקום האחרון שבו מצא סגמנט מתאים, במקום בכל פעם להתחיל מהתחלה. הביצוע שלו פחות טוב מ-FIRST FIT.

<u>BEST FIT</u> – מנהל הזיכרון מבצע סריקה על כל הרשימה במטרה למצוא את הסגמנט הקטן ביותר שמתאים לאחסון התהליך, זאת במטרה למנוע "שבירה" של סגמנט גדול ולמנוע בזבוז מקום. BEST FIT מנסה למצוא מקום מתאים לגודל הממשי שלו התהליך זקוק. נתבונן באיור 7-3 שוב. אם זקוקים לבלוק בגודל 2, FIRST FIT יקצה מקום בחור מספר 5, אולם BEST FIT יקצה מקום בחור מספר 18. BEST FIT איטי יותר מ-FIRST FIT היות והוא סורק את כל הרשימה בכל פעם שמגיע תהליך. ובאופן מפתיע הוא מסיים עם יותר מקום מבוזבז בזיכרון מאשר FIRST FIT או NEXT FIT היות והוא מנסה למלא את הזיכרון ביחידות קטנות מידי, וחסרות שימוש. FIRST FIT מייצר חורים גדולים יותר בממוצע.

<u>WORST FIT</u> בניגוד ל-BEST FIT תמיד לוקח את הבלוק הפנוי הגדול ביותר שבנמצא, כדי שגודל הזיכרון שיישאר לאחר השבירה יהיה גדול מספיק לשימוש. הדבר בא במטרה למנוע מצב בו לאחר "שבירה" של סגמנט קטן יישאר סגמנט קטן עוד יותר שלא יוכל להכיל אף תהליך. אלגוריתם זה אינו טוב דיו.
את כל ארבעת האלגוריתמים הללו ניתן לשפר בזמן על ידי אחזקת שתי רשימות נפרדות, לסגמנטים תפוסים, ולסגמנטים פנויים. אולם העלות היא ביותר מורכבות האלגוריתמים, והאטה כאשר משחררים סגמנט היות ויש להעביר אותו מרשימת התהליכים לרשימת הפנויים.
אם רשימת הסגמנטים הפנויים תמוין לפי גודל BEST FIT ישופר בזמן. כאשר הוא מחפש חור קטן דיו המתאים לתהליך זמן החיפוש יתקצר. באופן זה ביצועי BEST FIT ו- FIRST FIT יהיו מהירים באופן שווה, ו- NEXT FIT לא יהיה שימושי יותר.
בשיטת 2 הרשימות הנפרדות במקום להחזיק מבנים שונים של DATA כדי לשמור את רשימת הסגמנטים הפנויים, כמו שנעשה באיור 7-3(c), המילה הראשונה בכל מקום פנוי תכיל את גודלו, והמילה השניה תצביע על הכניסה הבאה, וכך לא נזדקק ל-3 מילים.
<u>QUICK FIT</u> - מנהל הזיכרון מחזיק רשימות נפרדות לגדלים שכיחים של תהליכים. (קיימת רשימה של מצביעים לכל סוג של רשימת סגמנטים).
לדוגמא, תהיה לו טבלה עם N כניסות, כאשר הכניסה הראשונה היא מצביע לתחילת הרשימה שבה מקומות פנויים בני 4K הכניסה השניה תצביע על תחילת רשימה של

מקומות פנויים בני 12K, וכך הלאה. מקומות פנויים בני 21K יוכלו להישמר ברשימה של 20K, או ברשימה מיוחדת של מקומות פנויים בגודל לא זוגי. היתרון הוא שמציאת מקום פנוי בגודל מתאים, הוא מהיר, אולם החיסרון הוא כבכל האלגוריתמים שבהם ממינים מקומות פנויים לפי גודל

עמוד 86

הוא שכאשר תהליך מסתיים או מוחלף קשה למצוא את שכניו כדי לראות האם ניתן להתמזג אתם. אם לא ניתן להתמזג, אזי במהרה נקבל רסיסי זיכרון, ומספר רב של חורים קטנים וחסרי תועלת.

אם נוריד את ההנחה המוקדמת שלא ידוע דבר מראש לגבי התפלגות הגדלים המבוקשים, ואורך חיים של תהליך, אזי אלגוריתמים נוספים יתאימו. נתאר כמה אפשרויות.

3.2.4 ניהול הזיכרון בשיטת הזוגות – BUDDY SYSTEM

בקטע הקודם ראינו ששמירת החורים ברשימה אחת או יותר ממוינים לפי גודל, שיפרה את ביצוע ההקצאה, אולם האטה את שחרור הזיכרון היות והיה צריך לסרוק את כל רשימות החורים כדי למצוא את שכני הסגמנט ששוחרר. שיטת הזוגות היא אלגוריתם לניהול הזיכרון המנצל את העובדה שהמחשב משתמש במספרים בינארים למיעון כדי לזרז את המיזוג של חורים צמודים כאשר תהליך מסתיים או מוחלף. להלן אופן הפעולה:

מנהל הזיכרון שומר רשימה של בלוקים פנויים בגדלים בחזקות של 2 (1,2,4,8,16), כך עד לגודל הזיכרון. עם זיכרון בגודל 1 מגה לדוגמא יהיה 21 בלוקים כאלו במתחם בין 1 בית ל-1 מגה. שאר הרשימות ריקות. מצב תחילתי של הזיכרון מודגם באיור 9-3.

נניח שתהליך בגודל 70K נכנס לזיכרון כשהוא ריק. היות והחורים הם בחזקות של 2, הבלוק המתאים יהיה בין 128K. אי לכך מתחילים לחלק את הזיכרון לזוגות (BUDDIES). כלומר בכל שלב מחלקים ב- 2 עד שמגיעים לבלוק של 128K. שאר הבלוק מבוזבז. כלומר בזיכרון בגודל 1 מגה יחולק הזיכרון ל- 2 בלוקים בני 512K כל אחד. האחד בכתובת 0, והשני בכתובת 512K. הבלוק בכתובת 0 יפצל את עצמו שוב לזוגות של בלוקים. זאת עד לבלוקים בגודל של 128K, והבלוק בכתובת 0 יוקצה לתהליך.

עמוד 87

לאחר מכן מגיע תהליך בגודל של 35K . גודל הבלוק בחזקת 2 המתאים הוא 64K. ולכן אנו מחלקים את הבלוק בין 128K לשני זוגות, בני 64K אחד בכתובת 128K והשני בכתובת 192K. הבלוק בכתובת 128K יוקצה לתהליך, והוא מסומן B באיור 9-3. גודל התהליך השלישי שנכנס הוא 80K.

כעת נבדוק מה קורה כאשר מוחזר בלוק. נניח שבלוק A שהוא בגודל 128K התפנה כעת. אזי הוא עובר לרשימת הפנויים בגודל 128K בשלב זה. כעת זקוקים לבלוק בגודל 60K . נערכת בדיקה למציאת הגודל הקרוב, שהוא 64K, והוא נמצא בכתובת 192K.

כעת בלוק B מוחזר. בשלב זה, יש לנו בלוק בן 128K בכתובת 0, ובלוק פנוי בן 64K בכתובת 128K. כל מיזוג אינו אפשרי עדיין. היות ופעולת המיזוג יכולה לפעול על בלוקים פנויים צמודים, שיש להם אותו אב.

כאשר יוחזר בלוק D נוכל לשחזר בלוק בן 256K בכתובת 0. לבסוף כאשר מוחזר בלוק D אנו חוזרים למבנה המקורי של 1 מגה.

יתרון השיטה על שיטות אחרות שממינות בלוקים לפי גודל, אולם לא בהכרח בכתובות שהם כפולות הגודל הוא שהמיזוג פשוט. כאשר בלוק בגודל 2 בחזקת K מוחזר, מנהל הזיכרון מחפש רק ברשימת הפנויים בגודל 2 בחזקת K. באלגוריתמים האחרים המאפשרים חלוקה לגדלים שרירותיים, יש לסרוק את כל רשימות הפנויים. ולכן החיפוש בשיטת הזוגות הוא מהיר.

החיסרון הוא שהאלגוריתם אינו יעיל במונחים של ניצול הזיכרון. הבעיה נובעת מהעובדה שיש לעגל את כל הבקשות לזיכרון לגדלים של חזקות 2. תהליך בן 35K יקבל בלוק בן 64K. 29K הנוספים מבוזבזים. צורה זו של בזבוז נקראת INTERNAL FRAGMENTATION (חלוקה פנימית) היות והזיכרון המבוזבז נמצא בתוך הסגמנט המוקצה. באיור 3-5 החורים נמצאים בין הסגמנטים, ואין מקום מבוזבז בתוך הסגמנטים. צורת בזבוז זו נקראת EXTERNAL FRAGMENTATION

(חלוקה חיצונית) אוCHECKERBOARDING לוח שחמט.

3.2.5 הקצאת מקום להחלפה

האלגוריתמים שהוצגו לעיל הם לשמירת מעקב ההקצאות בזיכרון הראשי כך שכאשר תהליכים נכנסים, המערכת יכולה למצוא מקום עבורם. בכמה מערכות כאשר תהליך נמצא בזיכרון, לא מוקצה לו שטח דיסק כאשר יש להורידו מהזיכרון, יש להקצות לו מקום על הדיסק. כאשר בכל הורדה התהליך יכול להישמר בחלק אחר בדיסק.
האלגוריתמים לתחזוקת מקום להחלפות אותם אלו שבזיכרון המרכזי.
במערכות אחרות, כאשר נוצר תהליך, מוקצה לו מקום שלו בלבד להחלפה על הדיסק. בכל פעם שתהליך יוצא מהדיסק, כשהוא חוזר, תמיד מוקצה לו אותו מקום לאחסון בדיסק, במקום להתאחסן במקום אחר בדיסק בכל פעם. כאשר התהליך מסתיים, שטח הדיסק שהוקצה לו אינו שייך לו עוד.

עמוד 88

היות ושטח הדיסק גדול – הקצאה קבועה חוסכת את זמן ההקצאה מחדש וזאת בניגוד לבעייתיות של שטח זיכרון ששם מקום ההקצאה משתנה.

3.2.6 ניתוח ההחלפה בזיכרון

רשימת הפנויים, ומפת הסיביות מובילים אותנו לצורה של חלוקה חיצונית שקלה לניתוח. תארו לכם שיש להחליט בכל רגע כמה זיכרון מבוזבז כעת בזיכרון. ניתן לסרוק את הזיכרון ולסמן חורים ותהליכים. אולם אם מקומות פנויים סמוכים היו מתמזגים מספר החורים היה קטן ממספר הסגמנטים. היחס בין מקומות פנויים לתהליכים יכול להיות בהתאם לניתוחים הבאים: נניח שתהליך ממוצע נמצא באמצע הזיכרון. במשך שהותו בזיכרון, חצי מהפעולות על הסגמנט מעליו יהיו הקצאות זיכרון לתהליכים וחצי פעולות של החזרת מקום של תהליך שהסתיים. לפיכך, מחצית מהזמן יש לו תהליך שכן למעלה, ומחצית מהזמן יש לו חור בשכן העליון שלו. ובממוצע צריכים להיות חצי חורים ממספר התהליכים בזמן נתון. ובמילים אחרות אם יש N תהליכים בזיכרון אזי מספר החורים הוא N\2 . התוצאה נקראת FIFTY PERCENT RULE.
חוק ה- 50% מתחיל בחוסר סימטריה קיצוני בין חורים לתהליכים. שני חורים סמוכים מאוחדים לחור אחד. תהליכים סמוכים לא מתמזגים. המנגנון באופן אוטומטי מקטין את מספר החורים.
תוצאה שימושית אחרת נקראת חוק הזיכרון הבלתי מנוצל. יהי F האחוז בזיכרון שמבוזבז על חורים. S יהיה הגודל הממוצע של N תהליכים, ו- KS יהיה הגודל הממוצע של חור כאשר K גדול מ- 0. עם זיכרון בגודל של M בתים N\2 החורים תופסים M-NS בתים. באופן אלגברי כלומר N\2 X KS = M-NS
M = NS(1 + K\2). לאחר חישוב מתקבלת התוצאה הסופית F= K\(K2+) .
כאשר K הוא גודל החור ביחס לגודל תהליך (בממוצע).
לדוגמא, אם גודל החורים הוא חצי מגודל התהליכים אזי K=1\2, ו- 20% מהזיכרון יבוזבז על חורים. אם נקטין את גודל החור הממוצע ל- 1\4 מגודל התהליך הממוצע, למשל, על ידי שימוש באלגוריתם BEST FIT במקום ב- FIRST FIT, הבזבוז יקטן ל- 11%. כל עוד גודל החורים הוא חלק ניכר מגודל התהליך הממוצע, חלק משמעותי מהזיכרון יבוזבז.

3.3 זיכרון מדומה

לפני שנים רבות אנשים ניצבו לראשונה מול תוכניות שהיו גדולות מידי מהזיכרון הפנוי. הפתרון שאומץ היה לחלק את התוכנית לחלקים, נקרא OVERLAYS – קיפול. לפי שיטה זו חלק 0 ירוץ תחילה, וכשהוא יסיים הוא יקרא לחלק הבא. כמה מערכות בשיטה זו היו מאוד מסובכות, היות ואפשרו לכמה חלקים לפעול במקביל בזיכרון. חלקי התוכנית אוחסנו בדיסק והועברו משם לזיכרון, ובחזרה על ידי מערכת ההפעלה. למרות שההחלפה בוצעה על ידי מערכת ההפעלה, חלוקת התוכנית לחלקים בוצעה על ידי המתכנת. הדבר היה משעמם וצרך זמן. ולא עבר זמן רב בטרם תוכנן אלגוריתם לביצוע

החלוקה על ידי המחשב – והוא נקרא זיכרון מדומה. הרעיון העומד מאחורי הזיכרון המדומה הוא שגודל תוכנית, כולל נתונים ומחסנית יכול לעלות על גודל הזיכרון הפיזי הפנוי. מערכת ההפעלה שומרת את חלקי התוכנית הפועלים בזיכרון המרכזי, והשאר נשמר בדיסק. לדוגמא, תוכנית בן 1 מגה יכולה לרוץ על מחשב בן 256K, על ידי בחירה זהירה איזה 256K ירוצו בכל רגע, כאשר חלקי התוכנית מועברות בין הדיסק והזיכרון בעת צורך.
זיכרון מדומה יכול לפעול במערכות רבות משתמשים. לדוגמא, 8 תוכניות בנות 1 מגה, יכולות לקבל שטח זיכרון של 256K כל אחת מסך זיכרון כולל של 2 מגה, כאשר כל תוכנית פועלת כאילו היא לבדה על מחשב עם זיכרון בגודל 256K. למעשה זיכרון מדומה וריבוי תוכניות מתאימים מאוד יחדיו. כאשר תוכנית ממתינה שחלק ממנה יוכנס מהדיסק, הוא ממתיק לקלט\פלט ולא יכול לרוץ, אזי המעבד יקצה משאבים בינתיים לתהליך אחר. מקבילות המערכת עולה וכך גם ניצול מרכיביה. הזמן הממוצע של ריצת כל התהליכים יורד.

3.3.1 דפדוף

רוב מנגנוני הזיכרון המדומה משתמשות בטכניקה הנקראת דפדוף.
בכל מחשב קיימים סטים של כתובות שהתוכניות יכולות לייצר (על ידי שימוש באינדקסים, אוגרים שונים ודרכים אחרות). כתובות אלו נקראות VIRTUAL ADDRESSES, כתובות מדומות, והן יוצרות את מרחב הכתובות הוירטואליות.
הוראות כגון MOVE REG,1000 מעתיקות את תוכן הזיכרון בכתובת 1000 לאוגר (או להפך, תלוי במחשב). במחשבים ללא זיכרון מדומה, הכתובת המדומה נשלחת ישירות לזיכרון והופכת למילה פיזית בזיכרון עם אותה כתובת, לקריאה וכתיבה. כאשר משתמשים בזיכרון מדומה, הכתובת המדומה לא נשלחת ישירות מהמעבד לזיכרון אלא היא נשלחת אל חלק הקרוי – MEMORY MANAGEMENT UNIT. זהו שבב או כמה שבבים שאחראי על מיפוי הכתובות הוירטואליות לכתובות פיזיות בזיכרון. שרטוט 3-10 עמוד 90. דוגמא כיצד מתבצע המיפוי מוצגת באיור 3-11.

עמוד 90

בדוגמא זו מחשב יכול לייצר כתובות בגודל 16 ביט, מכתובת 0 ועד 64K. אלה הכתובות הוירטואליות. אולם למחשב זה יש רק 32K של זיכרון פיזי, כך שלמרות שתוכניות בנות 64K יכולות להיכתב, הן אינן יכולות להיטען לזיכרון בשלמותן, ולרוץ. לכן כל התוכנית נשמרת בדיסק, וחלקים ממנה יוכלו לעבור לזיכרון.
מרחב הכתובות המדומה מחולק לחלקים הנקראים דפים. גם הזיכרון הפיזי, במקביל, נחלק ליחידות הנקראות PAGE FRAMES מסגרות דפים. הדפים והמסגרות תמיד באותו גודל. בדוגמא זו גודל הדפים הוא 4KB, אולם הוא יכול לנוע בין 512 בתים ל- 8K.
עם מרחב זיכרון מדומה בגודל 64K, וגודל פיזי של זיכרון בגודל 32K אנו מקבלים 16 דפים וירטואלים ו- 8 מסגרות. המעברים בין הזיכרון לדיסק הם תמיד ביחידות של דפים.
כאשר תוכנית מנסה להיכנס לכתובת 0, לדוגמא, תוך שימוש בהוראה MOVE REG,0 הכתובת המדומה 0 נשלחת ליחידת ניהול הזיכרון (MMU). הוא בודק שהכתובת המדומה הופכת לדף 0, (בכתובת 0 עד 4095) אשר בהתאם למפה זהו מסגרת 2 (בכתובת 8192 עד 12287).לפיכך הוא משנה את הכתובת ל- 8192 . הזיכרון לא יודע על פעולת ה-MMU, הוא רואה רק בקשות לקריאה או כתיבה בכתובת 8192, שאותן הוא מכבד.
לפיכך, ה-MMU מיפה את כל הכתובות הוירטואליות בין 0 ל- 4095 לכתובות פיזיות בין 8192 ל- 12287. בדומה ההוראה MOVE REG,8192 תשונה באופן מעשי ל-MOVE REG,24576 היות והכתובת הוירטואלית 8192 נמצאת בדף וירטואלי מספר 2 ודף זה ממופה למסגרת פיזית מספר 6 (כתובות פיזיות בין 24576 ל- 28671). כלומר ה-MMU מקבל כתובת וירטואלית, משייך אותה לדף וירטואלי ומחשב מהו ה- OFFSET מתחילת הדף לכתובת הוירטואלית. לאחר מכן לפי שיוך דף וירטואלי לדף פיזי, מגיע אל הדף הפיזי ומחשב את הכתובת הפיזית לפי OFFSET + כתובת תחילת הדף הפיזי.

עמוד 91

היכולת למפות 16 דפים וירטואליים לתוך 8 דפים פיזיים על ידי ה-MMU עדיין אינה פותרת את הבעיה שמרחב הכתובות המדומות גדול מהזיכרון הפיזי. עדיין נותרים יותר דפים וירטואליים מאשר דפים פיזיים. כלומר היות ויש לנו 8 מסגרות דפים פיזיים בלבד רק 8 דפים וירטואליים יוכלו להיות ממופים לדפים פיזיים, השאר אינם ממופים.
בחומרה קיים PRESENT\ABSENT BIT ביט לכל כניסה המורה האם הדף ממופה או לא.
מה קורה כאשר התוכנית פונה לכתובת וירטואלית שאינה ממופה, לדוגמא על ידי ההוראה MOVE REG,32780 שזהו הבית ה- 12 בדף וירטואלי 8 (המתחיל בכתובת 32768)? ה-MMU מאתר את התקלה, וגורם למעבד ליפול לתוך מלכודת הנקראת PAGE FAULT. במצב זה מערכת ההפעלה מחזירה לדיסק דף פיזי , רושמת את תוכנו בדיסק ומביאה במקומו את הדף הרצוי, השינויים הנדרשים במיפוי ה- MMU נעשים, וההוראה מותחלת מחדש.
לדוגמא, אם מערכת ההפעלה מחליטה לפנות את הדף הממוסגר 1, היא תטען את דף וירטואלי 8 לכתובת פיזית 4K ותבצע 2 שינויים במפת ה-MMU. תחילה היא תסמן את כניסת הדף הוירטואלי מספר 1 כלא ממופה, כדי ללכוד כל כניסה עתידית למרחב כתובות מדומה בין 4K ל- 8K. ואז היא תחליף את הצומת בכניסה לדף וירטואלי 8 למספר 1, כך שכאשר תבוצע מחדש ההוראה שנלכדה, היא תמפה את מרחב הכתובות 32780 אל הכתובת הפיזית 4108.

עמוד 92

כיצד פועלת יחידת ניהול הזיכרון (MMU)? ומדוע בחרנו בדפים בגדלים של חזקות ? באיור 3-12 אנו רואים דוגמא של כתובת וירטואלית, 8196 (0010000000000100 בבינארי), הממופה על ידי ה-MMU. 16 הביטים הללו מחולקים כך ש- 4 סיביות ראשונות מייצגות דף וירטואלי, ו- 12 הסיביות הבאות את ה- OFFSET. עם 4 סיביות עבור מספר דף אנו יכולים לייצג (2 בחזקת 4) 16 דפים וירטואלים, ועם 12 סיביות, עבור ה-OFFSET ניתן לייצג את כל 4096 הסיביות שבדף (2 בחזקת 12).
מספר הדף משמש כאינדקס בטבלת הדפדוף, ומספר המסגרת נקבע בהתאם לדף הוירטואלי. טבלת הדפדוף מייצגת 16 דפים כאשר הסיבית האחרונה מתוך הארבע היא סיבית הנוכחות, אם סיבית הנוכחות (PRESENT\ABSENT) היא 0, (כלומר הדף אינו ממופה, ונמצא בדיסק) אזי נגרם מצב מלכודת למערכת ההפעלה. אם הסיבית היא 1, מספר המסגרת נמצא בטבלת הדפים, מיוצג על ידי 3 הסיביות הראשונות , והוא מועתק ל-OUTPUT REGISTER יחד עם 12 סיביות ה-OFFSET . יחד מעביר האוגר את כל 15 הסיביות של אל הזיכרון ככתובת פיזית. (ברור יותר להתבונן באיור 3-12).
שוב, לקבלת הכתובת הפיזית : סיביות ה-OFFSET מועתקות כפי שהן, ומצד שמאל מוסיפים את שלוש הסיביות הרשומות באינדקס הדף הוירטואלי בטבלה (הסיביות לייצוג דף פיזי).

3.3.2 טבלת הדפים

באופן תאורטי מיפוי הכתובות המדומות לכתובות פיזיות נעשה כמתואר בעמוד

עמוד 93

הקודם. הכתובת הוירטואלית מחולקת למספר דף וירטואלי (הסיביות היותר משמעותיות של הכתובת), ול- OFFSTE. מספר הדף הוירטואלי משמש כאינדקס בטבלת הדפים על מנת למצוא כניסה עבור הדף. מטבלת הדפדוף, נמצא מספר המסגרת. מספר המסגרת משורשר לראשית ה- OFFSET, מחליף את מספר הדף הוירטואלי, כדי לייצג כתובת פיזית שיכולה להישלח לזיכרון.
הסיבה לטבלת הדפדוף היא למפות את הדפים הוירטואלים למסגרות. ניתן לומר שטבלת הדפדוף היא פונקציה, עם מספר הדף הוירטואלי כארגומנט, ומסגרת פיזית כתוצאה.
למרות התיאור הפשוט לכאורה ישנם שני נושאים חשובים שיש להתמקד בהם :

1. טבלת הדפדוף יכולה לתפוס מקום רב. בייחוד לאור העובדה שכיום מחשבים משתמשים בכתובות של 32 סיביות או 64 סיביות, ניתן להגיע למליון דפים ועבורם צרים כניסות בטבלה (כלומר מיליון כניסות). כמו כן יש לזכור שיש לשמור טבלה שכזו <u>עבור כל תהליך.</u>

2. זמן – המיפוי חייב להיות מהיר. וזוהי תוצאה של העובדה שהמיפוי המדומה אל הפיזי חייב להעשות בכל התייחסות לזיכרון. להוראה טיפוסית יש מילת הוראה ולעיתים קרובות גם אופרנד. כתוצאה מכך יש לעשות פניה אחת או שתיים ולעיתים יותר אל טבלת הדפדוף לכל הוראה. לכן סריקה בטבלה צריכה לקחת לכל היותר כמה ננו-שניות אחרת יהפוך המיפוי לצוואר בקבוק.

הצורך במפות גדולות ומהירות, הוא אילוץ משמעותי בבנית מחשבים. למרות שהבעיה רצינית ביותר במחשבים הטובים ביותר. זהו נושא שבו הביצועים הם קריטיים. בפרק זה נעסוק בפרטי עיצוב טבלת הדפדוף, ונראה כמה פתרונות חומרה שהיו בשימוש.

העיצוב הפשוט ביותר (לפחות באופן תאורטי), הוא טבלה יחידה הבנויה ממערך של אוגרים מהירים, עם כניסה אחת לכל דף וירטואלי, האינדקס הוא מספר הדף הוירטואלי. כאשר מתחיל תהליך מערכת ההפעלה טוענת את הרגיסטרים עם טבלת הדפדוף של התהליך, הנלקחת מהעתק השמור בזיכרון הראשי. במשך ביצוע התהליך, אין צורך בהפניות לטבלת הדפדוף. היתרון הוא בפשטות וכן שאין פניות לזיכרון במשך המיפוי. החיסרון הוא שהוא יקר בפוטנציה (אם הטבלה גדולה). וכן שיש לטעון בכל פעם את הטבלה, מה שגם יכול לפגום בביצועים.

מצד שני, טבלת הדפדוף יכולה לשהות בזיכרון המרכזי. החומרה הנדרשת היא אוגר אחד בלבד המצביע על תחילת הטבלה. החיסרון הוא שנדרשת פניה אחת או שתיים לכל הוראה, בעת ביצוע התהליך. מסיבה זו גישה זו כמעט ולא בשימוש.

<u>עמוד 94</u>

<u>טבלת דפדוף שכבתית – MULTILEVEL PAGE TABLE</u>

הבעיה היא שטבלאות ענקיות נמצאות כל הזמן בזיכרון, טבלת הדפדוף השכבתית באה לפתור בעיה זו.

למעשה התהליכים מנצלים בו זמנית רק חלק זעיר ממרחב הכתובות הוירטואלי. אי לכך משתמשים באותו רעיות העומד ביסוד מנגנון הדפדוף עצמו, וכך גם טבלת הדפדוף תכיל בו זמנית רק את המידע הנחוץ לתרגום מרחב מענים קטן ביותר. שאר המידע נשאר על הדיסק ומובא לפי הצורך. בדיוק כפי שהדפים עצמם שוכנים ברובם המכריע בדיסק ומובאים לזיכרון לפי הצורך בהם. לדוגמא קיים המנגנון של TWO-LEVEL PAGE TABLE.

לפי מנגנון זה מחלקים את הטבלה הגדולה המכילה את כל הדפים הוירטואליים, לחלקים, כאשר בונים עוד טבלה שאחראית על הטבלה הראשונה ורק היא נמצאת כל העת בזיכרון, הטבלה הגדולה נמצאת בדיסק.

כעת כתובת נחלקת ל-שלושה חלקים ראה איור 13-3. באיור יש לנו כתובות בנות 32 סיביות שנחלקות : 10 ביטים המשמשים גישה לטבלה הראשונה המוחזקת בזיכרון. לטבלה זו יש אינדקסים לטבלה הגדולה המוחזקת בדיסק. 10 ביטים נוספים שהם גישה לטבלה השניה המוחזקת בדיסק, ו- 12 ביטים עבור ה-OFFSET.

<u>עמוד 95</u>

הדוגמא באיור 13-3 מציגה כתובות בנות 32 ביט המחולקות ל- 3.

כמו כן נניח שתהליך זקוק ל- 12 מגה. בתחתית הזיכרון ישנם 4 מגה עבור תוכן התוכנית, מעל 4 מגה עבור הנתונים, למעלה 4 מגה עבור המחסנית, וביניהם חור שאינו בשימוש.

איור 13-3 מציג לנו כיצד פועלת טבלה דו שכבתית. משמאל ישנה טבלה עם 10הסיביות היותר משמעותיות, עם 1024 כניסות. כאשר הכתובת הוירטואלית מוצגת ב-MMU, הוא לוקח את 10 הביטים הראשונים ומשתמש בהם כאינדקס לשכבה גבוהה של טבלת הדפדוף. כל אחת מ- 1024 הכניסות מייצגת 4 מגה היות וכל מרחב הכתובות המדומות בן 4 גיגה כווץ לתוך 1024 הכניסות.

הכניסה הממוקמת על ידי אינדקס בשכבה העליונה, מצביעה על כתובת או מספר מסגרת בשכבה השניה. כניסה 0 של השכבה העליונה מצביעה על הטבלה עבור התוכנית. כניסה 1 מצביעה על הנתונים, וכניסה 1023 מצביעה על המחסנית. הכניסות האחרות אינן בשימוש. טבלה מספר 2 משמשת כאינדקס לשכבה שניה של הטבלה, ומאפשרת למצוא את מספר המסגרת עבור הדף עצמו.

ניקח כדוגמא כתובת וירטואלית בת 32 ביט 0X00403004 (4,206,596 דצימלי) המכיל 12292 בתים של DATA. כתובת זו מחולקת כך : PT1=1 (טבלה 1). ,PT2=3 (טבלה 2) ו-

OFFSET=4. ה- MMU משתמש תחילה בטבלה הראשונה PT1 ומקבל את כניסה מספר 1, אשר מייצגת את תחום הכתובות 4 מגה ועד 8 מגה. אזי הוא משתמש בטבלה 2 בכניסה 3, אשר מייצגת את תחום הכתובות
12288 ועד 16383 וזהו חלק מהתחום של 4 מגה. הכניסה מכילה את מספר המסגרת של הדף המוכל בכתובת 0X0403004. אם הדף לא נמצא בזיכרון אזי סיבית הנוכחות תהיה 0, ותגרום ל- PAGE FAULT. אם הדף בזיכרון, מספר המסגרת שנלקח מטבלה 2 משורשר משמאל ל-OFFSET על מנת ליצור כתובת פיזית. כתובת זו נשלחת לזיכרון.
המעניין הוא שלמרות שמרחב הכתובות מכיל למעל למיליון דפים, נזקקים רק ל- 4 דפי טבלאות: טבלה 1 שבשכבה העליונה, טבלה 2 שבשכבה השניה עבור 0 עד 4M, 4M עד 8M, ו- 4M העליונים. סיבית הנוכחות ב- 1021 כניסות של טבלה 1 מכוונת ל- 0, אוכפת PAGE FAULT. אם מערכת ההפעלה שמה לב שתהליך מנסה לפנות לכתובת בזיכרון, שהוא לא אמור לפנות אליה, היא תנקוט באמצעים המתאימים, כמו שליחת איתות, או הפסקתו. בדוגמה זו בחרנו מספרים עגולים לגדלים שונים, ובחרנו PT1 שווה ל- PT2, אולם במציאות, ערכים אחרים אפשריים, כמובן.
מנגנון טבלת הדפדוף הדו-שכבתית יכולה להתרחב לשלוש, ארבע ויותר שכבות. שכבות נוספות מעניקות יותר גמישות, אולם בספק אם המורכבות הנוספת מצדיקה יותר משלוש שכבות.
נעבור עתה לפרטי כניסה אחת בטבלת הדפדוף. התכנון המדויק של הכניסה משתנה אולם אינפורמציה מסוימת הכרחית בכל המחשבים. איור 14-3 מדגים כניסה בטבלת דפדוף.
הגודל משתנה ממחשב למחשב, אולם הגודל הנפוץ הוא 32 ביט. השדה החשוב ביותר הוא מספר המסגרת. לאחר הכל, המטרה של מיפוי הדף היא למקם את הערך הזה. אחר כך ישנה סיבית הנוכחות. אם היא 1 הכניסה זמינה וניתנת לשימוש. אם היא 0, הדף הוירטואלי שכניסה זו שייכת לו, לא נמצא כרגע בזיכרון. כניסה לטבלה, עם סיבית נוכחות 0 גורמת ל- PAGE FAULT.

עמוד 96

שדה ההגנה (PROTECTION BITS) מראה מהו סוג הכניסה המותר. בצורה הפשוטה ביותר, השדה מורכב מביט אחד, כאשר 0 הוא עבור READ\WRITE , ו- 1 הוא עבור קריאה בלבד. בצורה מתוחכמת יותר השדה בן שלושה ביטים, כאשר כל אחד מהם הוא עבור הרשאת כתיבה, קריאה, וביצוע הדף.
סיביות השינוי וההתיחסות עוקבות אחר השימוש בדף. כאשר כותבים לדף, החומרה מייד מדליקה את סיבית השינוי (MODIFIED BIT). משתמשים משתמשים בסיבית זו כאשר מערכת ההפעלה מחליטה לדרוש בחזרה את מסגרת הדף. אם הדף המוכל במסגרת שונה, יש לכתוב את הדף חזרה בדיסק, אחרת, אין צורך לעשות דבר היות וההעתק בדיסק עדיין תקף.
סיבית ההתייחסות (REFERENCED BIT) נדלקת כאשר ישנה פניה לדף, לקריאה או לכתיבה. הסיבית נועדה לעזור למערכת ההפעלה לבחור דף שיש לפנות, כאשר נגרם PAGE FAULT. דפים שלא היו בשימוש הם מועמדים טובים יותר לפינוי, ולסיבית זו תפקיד חשוב במספר אלגוריתמים להחלפת דפים, שנלמד מאוחר יותר בפרק זה.
הסיבית האחרונה, שהיא סיבית הסימון לזיכרון מטמון מהיר (CACHING DISABLED). היא חשובה עבור דפים שמופו לתוך רגיסטרים, יותר מאשר לזיכרון. יש לבקש הבהרה בכיתה עבור סיבית זו.
מחשבים שיש להם מרחב נפרד עבור קלט\פלט ולא משתמשים במפה בזיכרון עבור קלט\פלט, לא זקוקים לסיבית זו.
נציין שהכתובות בדיסק המשמשות לשמירת דף אשר לא נמצא בזיכרון, אינן חלק מטבלת הדפדוף. הסיבה היא פשוטה. טבלת הדפדוף מכילה רק את המידע שהחומרה צריכה כדי לתרגם כתובת מדומה, לכתובת פיזית. מידע שמערכת ההפעלה זקוקה לו כדי לטפל ב- PAGE FAULT נשמר בטבלאות תכנותיות בתוך מערכת ההפעלה.

עמוד 101

3.3.4 זיכרון אסוציאטיבי

מתיאור מערכת הדפדוף עולה כי העלות העיקרית במונחים של זמן נובעת מן הצורך בגישה לטבלת הדפדוף לפני כל פנייה לזיכרון.
בדרך כלל טבלת הדפדוף נשמרת בזיכרון. בפוטנציה, לתכנון הטבלה ישנה השפעה גדולה על ביצועיה. נניח למשל הוראה שמעתיקה אוגר אחד למשנהו.
בהעדר דפדוף, הוראה זו מבצעת פניה אחת לזיכרון, להביא את ההוראה. עם דפדוף יש צורך בפניה נוספת כדי להיכנס לטבלת הדפדוף. היות ומהירות הביצוע מושפעת מהמהירות שבה יכול המעבד להוציא הוראות ונתונים מהזיכרון. עם דפדוף ב- 68030 מהירות הביצוע היא רק 20% מהמהירות ללא דפדוף. בתנאים אלו לא ניתן להשתמש בדפדוף.

<u>עמוד 102</u>

הפתרון מבוסס על ההפשטה שרוב התוכניות נוטות לפנות מספר רב של פניות למספר קטן של דפים, ולא לכל הדפים באופן שווה.
לפיכך רק חלקים קטנים מכניסות הטבלה נמצאות בתדירות רבה, השאר כמעט ולא בשימוש. יישום הפתרון הוא על ידי חומרה. הוסיפו אמצעי חומרה קטן למחשב שתפקידו לתרגם כתובות וירטואליות לכתובות פיזיות, ללא צורך בגישה לטבלת הדפדוף.
אמצעי זה נקרא זיכרון אסוציאטיבי (ASSOCIATIVE MEMORY או לעיתים TRANSLATION LOODASIDE BUFFER). הוא מודגם באיור 20-3.

<u>עמוד 103</u>

האמצעי נמצא בדרך כלל ב- MMU ומכיל מספר קטן של כניסות, 8 בדוגמא זו אולם לעיתים נדירות יותר מ- 32. כל כניסה מכילה אינפורמציה לגבי דף אחד, את מספר הדף הוירטואלי, את סיבית השינוי, סיבית ההגנה, והמסגרת הפיזית שבה ממוקם הדף. שדות אלו מקבילות למידע שבטבלת הדפדוף. סיבית נוספת מלמדת האם הכניסה פנויה.
בדוגמא, התהליך נמצא בלולאה שמשתרעת על דפים וירטואלים 19, 20, ו- 21 כך שכניסות אלו נמצאות בזיכרון האסוציאטיבי, אם קוד ההגנה עבור קריאה וביצוע. הנתונים שבשימוש נמצאים בדפים 129 ו- 130. עמוד 140 מכיל את נתונים המשמש לחישובי המערך. לבסוף המחסנית נמצאת בעמודים 860 ו- 861. אופן פעולת הזיכרון האסוציאטיבי:
כאשר מגיעה כתובת וירטואלית ל- MMU לתרגום, אמצעי החומרה בודק ראשית אם הדף הוירטואלי נמצא בזיכרון האסוציאטיבי, על ידי השוואה לכל הכניסות בעמודה של מספר דף וירטואלי. אם נמצא הדף ויש הרשאות מתאימות, מספר המסגרת הפיזי נלקח ישירות ללא צורך בגישה לטבלת הדפדוף. אם הדף נמצא אך ההרשאה אינה מתאימה יוצר PROTECTION FAULT.
המקרה המעניין הוא מה קורה כאשר מספר הדף המבוקש אינו נמצא בזיכרון האסוציאטיבי? ה- MMU מגלה את החוסר ומבצע גישה רגילה לטבלת הדפדוף ואז מחליף את אחת הכניסות בזיכרון האסוציאטיבי בדף המבוקש. הכניסה שמוחזרת אל טבלת הדפדוף מוחזרת על ידי כך שסיבית השינוי מועתקת מהזיכרון האסוציאטיבי את טבלת הדפדוף, שאר הערכים כבר שם. כאשר הזיכרון האסוציאטיבי טוען מטבלת הדפדוף כל הדפים נלקחים מהזיכרון.

<u>עמוד 104</u>

מספר הגישות לזיכרון המופחת כתוצאה מהזיכרון האסוציאטיבי קרוי יחס ההצלחה (HIT RATIO). כמה שיחס ההצלחה גבוה יותר כך טובים הביצועים. יחס של 100% אומר שכמעט ולא משתמשים בטבלת הדפדוף עצמה אלא רק בזיכרון האסוציאטיבי.
מאידך, בתהליך המכניס רשימות מקושרות ארוכות לדפים רבים, יחס ההצלחה ישאף ל- 0 והוא ישתמש בטבלת הדפדוף כמעט בכל פניה.
כדוגמא, נניח שלוקח 100 ננו-שניות להיכנס לטבלת הדפדוף, ו- 20 ננו-שניות להיכנס לזיכרון האסוציאטיבי. עם יחס הצלחה של 90%, זמן הכניסות הממוצע יהיה 28 ננו-שניות. ליחס הצלחה אחר, עם פרמטרים אלו, זמן הכניסות הממוצע מודגם בגרף באיור 21-3.
באופן כללי, הביצוע הממוצע תלוי בשלושה גורמים:

1. זמן הגישה לדף בטבלה.
2. זמן הגישה לזיכרון האסוציאטיבי.

3. יחס ההצלחה – HIT RATIO.

יחס ההצלחה תלוי במספר הממוצע של פניות לדפים ביחידת זמן כלשהי, ובגודל הזיכרון האסוציאטיבי. היות ומספר הפניות לזיכרון תלוי בהתנהגות התוכנית, דבר שאינו בשליטת מתכנני ה- MMU, הדרך היחידה שלהם להעלות את יחס ההצלחה הוא להגדיל את מספר הכניסות בזיכרון האסוציאטיבי. מיותר לציין, שהדבר מגדיל את העלות.
תפקוד הזיכרון האסוציאטיבי במערכת של ריבוי תהליכים:
לכל תהליך יש את טבלת הדפדוף והמיפוי שלו. כאשר תהליך רץ זה חיוני שהוא לא ינסה להשתמש בכניסות בזיכרון האסוציאטיבי של התהליך הקודם.
באופן כללי ישנם שני פתרונות –

1. כאשר מותחל תהליך חדש, לספק הוראת מכונה שמאתחלת את הזיכרון האסוציאטיבי, כלומר מנקה את כל תוכן הסיביות.
2. להוסיף לזיכרון אסוציאטיבי עם תהליך, שדה שיכיל קוד זיהוי וכן להוסיף רגיסטר חומרה המכיל את קוד הזיהוי של התהליך הנוכחי. בדרך זו החומרה יכולה להשוות לא רק את מספר הדף הוירטואלי, אלא גם את קודי הזיהוי של התהליך, ורק התהליך הנוכחי יורשה להשתמש בכניסות שלו. גישה זו צורכת חומרה נוספת אך חוסכת בזמן במעברי ה- CONTEXT SWITCH.

עמוד 105

דפדוף ללא טבלת דפים ZERO-LEVEL PAGING

התיאור של מערכת זו מעניין במיוחד, שכן הוא ממחיש את העובדה כי לעיתים קרובות מערכת פשוטה עשויה להניב תוצאה שקולה לזו שנותנת מערכת מורכבת, ובעלות נמוכה בהרבה.
מערכת MIPS-R2000 פיתחה את רעיון הזיכרון האסוציאטיבי. טבלת הדפדוף בוטלה.
במערכת זו יש מערכת קונבציונלית של כתובות בנות 32 ביט, ודפים בני 4K. כל כתובת וירטואלית בת 32 ביט מחולקת כך: 20 ביט עבור מספר הדף הוירטואלי, ו- 12 ביט עבור OFFSET. הכתובות הפיזיות מיוצגות גם כן ב- 32 ביט.
המעבד מכיל 64 כניסות של זיכרון אסוציאטיבי על chip CPU. כל כניסה כזו מחזיקה: מספר דף וירטואלי, מספר מסגרת פיזית, קוד זיהוי של התהליך, ועוד כמה ביטים כדגלים שונים (שרטוט 22-3). כאשר המעבד מספק כתובת וירטואלית, החומרה באופן אוטומטי משווה את הדף הוירטואלי לכל הכניסות שבזיכרון האסוציאטיבי בבת-אחת. אם נמצאה התאמה, התרגום מתבצע והכתובת הפיזית נשלחת אל הזיכרון.
עד כה אין שינוי. החלק המעניין הוא מה קורה כאשר אין התאמה? בשונה מהמחשבים האחרים שעליהם למדנו, החומרה לא פונה לטבלת דפדוף אלה למערכת ההפעלה. מערכת ההפעלה חייבת אז לקבוע איזה מספר דף וירטואלי דרוש, על ידי הסתכלות ברגיסטר חומרה מסוים. אז היא בוחרת כניסה בזיכרון האסוציאטיבי וטוענת לתוכו את הדף החדש ומריצה את ההוראה. כמובן אם הדף לא נמצא בזיכרון, אזי היא גם גורמת לפסיקת דף גם כן.
החלק שמייעל את העבודה הוא הגישה לגרעין הגורמת למערכת לעבוד במהירות. כלומר החלק השונה הוא שמצב מלכודת בגרעין (KERNEL TRAP) נגרם לא רק על ידי פסיקת דף, אלא גם בחוסר בזיכרון אסוציאטיבי. הסיבה לתכנון שונה זה היא לשמר את פשטות המחשב, כדי להגביר את מהירות הביצוע. המתכננים חשו שטיפול בחוסר בזיכרון, על ידי חומרה יתפוס יותר מידי מקום של צ׳יפים, שעדיף לשמרו לדברים אחרים.

טבלת דפים מהופכת INVERTED PAGE TABLE.

אמצעי זה מאפשר לעבוד עם מערכות זיכרון שבהן מרחב הכתובות הווירטואלי מגיע לסדרי גודל עצומים, שאינם מאפשרים שימוש פשוט בטבלאות הדפים שהשתמשנו בהן עד כה. הסיבה היא שמספר הדפים האפשרי הוא כה גדול עד שמן הנמנע לשמור את טבלת הדפים בזיכרון. ואמנם בשיטה זו קיימת גם טבלת דפים רגילה, אך מפאת גודלה היא נשמרת על הדיסק בלבד. מטבע הדברים פעילותה הופכת לאיטית ביותר, ולפיכך נרצה להשתמש בה מעט ככל האפשר – רק כאשר הדף המבוקש אינו נמצא בזיכרון הראשי.
טבלת הדפים המהופכת אמורה לסייע לנו בכך שהיא מייצגת את מצבו של הזיכרון הראשי.

החיפוש של דף נדרש מתבצע על פי הצעדים הבאים:

1. חיפוש בזיכרון האסוציאטיבי. במקרה של כישלון עוברים לשלב 2.
2. חיפוש בטבלת הדפים ההפוכה. אם הדף נמצא בזיכרון מעדכנים את הזיכרון האסוציאטיבי ובמבצעים את הפעולה הדרושה. אחרת – ממשיכים לשלב 3.
3. מחפשים את הדף הנדרש בטבלת הדפים באמצעות מספר גישות לדיסק שעליו היא מאוחסנת ומביאים את הדף לזיכרון. אם אין מקום פנוי בזיכרון, יש צורך להחליט איזה דף – אם בכלל – כבר ניתן לפינוי. בבעיה זו עוסק הסעיף הבא.

כדוגמא, נביט בשני טבלאות הדפדוף שבאיור 3-23. הטבלה השמאלית היא טבלה קונבנציונלית, לפי תהליך, הממוינת לפי מספר דף וירטואלי. החומרה מפיקה את מספר הדף הוירטואלי, מהכתובת הוירטואלית ומשתמשת בה כאינדקס לטבלת הדפדוף. עם דף וירטואלי K, נבחרת הכניסה ה- Kית, ומסגרת הדף שלה משמשת לבניית הכתובת הפיזית. שלא בדומה לטבלה השמאלית, הטבלה הימנית לא ממוינת על ידי כתובת וירטואלית.

במקום, הכניסה ה- I-ית מכילה מידע לגבי הדף שכרגע נמצא במסגרת I. מספר הכניסות בטבלה לכן שווה למספר המסגרות בזיכרון הפיזי, ללא תלות במספר הספים במרחב הכתובות הוירטואליות. טבלת הדפדוף המוצגת באיור 3-23 מימין נקראת טבלת דפדוף מהופכת.

משתמשים בטבלה זה תמיד בצרוף עם זיכרון אסוציאטיבי. בהצלחה, אין צורך בטבלה המהופכת. בכישלון, יש לחפש בטבלת הדפדוף כניסה שבה מספר הדף הוירטואלי זהה לדף הוירטואלי שנמצא עתה בכתובת בזיכרון.

ניתן לזרז את החיפוש על שיש טבלת HASH של כל כניסות טבלת הדפדוף. החיפוש יכול להתבצע או על ידי החומרה, או על ידי מערכת ההפעלה. אם החיפוש מתבצע על ידי תוכנה, חשוב שלא יתבצע לעיתים קרובות.

אם הדף שזקוקים לו לא נמצא בזיכרון, על מערכת ההפעלה לחפשו. למטרה זו יש צורך בטבלת דפדוף קונבנציונלית, אפילו אם היא תאוחסן בדיסק ולא בזיכרון. כמובן שאחסון הטבלה בדיסק משמעותו שיהיה צורך בפסיקת דף, כדי שניתן יהיה לפנות לדיסק כדי להביא את הדף הדרוש.

<u>אלגוריתמים להחלפת דפים</u>

כאשר מתרחשת פסיקת דף, על מערכת ההפעלה לבחור בדף להסרה מהזיכרון על מנת לפנות מקום לדף הדרוש. אם במהלך השהייה בזיכרון הדף שונה, יש לשמור עותק שלו בדיסק לפני שזורקים אותו מהזיכרון, אם לא שונה אין בכך צורך כי ההעתק בדיסק עדיין מעודכן. הדף החדש פשוט נכתב על הדף הישן.

אמנם ניתן לבחור באופן אקראי דף להסרה מהזיכרון, אולם ביצועי המערכת טובים יותר אם נבחר דף שלא בשימוש רב. אם הוא בשימוש רב קרוב לודאי שנצטרך בקרוב לטעון אותו שוב לזיכרון.

להלן אלגוריתמים להחלפת דפים המטפלים בבעיה זו.

<u>עמוד 108</u>

<u>3.4.1 אלגוריתם אופטימלי להחלפת דפים</u>

האלגוריתם זה הוא פשוט אולם בלתי אפשרי ליישום.

תאור האלגוריתם: כאשר מתרחשת פסיקת דף, נעשית בדיקה לכל הדפים שכרגע בזיכרון – עוד כמה הוראות עד שהדף יהיה שימושי. הדף שיהיה שימושי רק עוד מספר הוראות הגדול ביותר יהיה זה שיוסר מהזיכרון לתוך הדיסק. לאחד הדפים למשל יפנו בהוראה הבאה, לדפים אחרים יפנו נניח בעוד 10, 100 או אולי 1000 הוראות. כל דף יסומן לפי מספר ההוראות שיש לבצע לפני הפניה לדף הזה.

החיסרון באלגוריתם היא שאינו מציאותי. בזמן פסיקת דף, אין למערכת ההפעלה אפשרות לדעת לאיזה מהדפים תהיה הפניה הבאה. אם נריץ את התוכנית בסימולטור ונרשום את כל הפניות לדפים, אזי ניתן יהיה ליישם את האלגוריתם, בהרצה השניה, תוך שימוש ברשימות שאספנו בהרצה הראשונה.

בדרך זו ניתן להשוות את ביצועי האלגוריתמים האחרים עם הטוב ביותר האפשרי. אם מערכת ההפעלה משיגה ביצועים גרועים באחוז אחד בלבד מהאלגוריתם האופטימלי, ניתן יהיה לרכז מאמצים בשיפור הביצוע.

כדי להימנע מבלבול, נבהיר שברישום פניות לדפים, הכוונה היא לתוכנית אחת שנמדדת. האלגוריתם האחר משווה ביצועים לגבי אותה תוכנית. אמנם אלגוריתם זה טוב למדידת ביצועי אלגוריתמים אחרים, הוא אינו מעשי.

3.4.2 אלגוריתם הדף העתיק – NRU

על מנת לאפשר למערכת ההפעלה לאסוף נתונים מועילים לגבי איזה דפים בשימוש ואילו לא, לרב המחשבים, עם זיכרון וירטואלי, יש שתי סיביות סטטוס בכל דף. R נדלקת כאשר יש פניה לדף לקריאה או לכתיבה.

עמוד 109

M נדלקת כותבים בדף (סיבית השינוי). הסיביות נמצאות בכל כניסה בטבלת הדפדוף. חשוב להבין שיש לעדכן סיביות אלו בכל פניה לזיכרון, ולכן חיוני שהן תופעלנה על ידי החומרה. ברגע שסיבית נדלקת ל- 1, היא נשארת 1 עד אשר מערכת ההפעלה מאפסת אותה על ידי תוכנה. אם לחומרה אין את הסיביות הללו, ניתן, לבצע את הסימולציה הבאה: כאשר תהליך מתחיל, כל הכניסות בטבלת הדפדוף שלו מסומנות כלא בזיכרון.
כאשר יש פניה לדף, מתרחשת פסיקת דף. מערכת ההפעלה מדליקה אז את R, ומשנה את הכניסה בטבלה, כדי להצביע על הדף הנכון, במצב READ ONLY, וממשיכה את ביצוע ההוראה.
אם כותבים לדף, פסיקת דף נוספת מתרחשת, כדי לאפשר למערכת ההפעלה להדליק את M ביט, ולשנות את מצב התהליך ל- READ\WRITE.
ניתן להשתמש בביטים R ו- M על מנת לתכנן את האלגוריתם הפשוט הבא:
כאשר תהליך מאותחל, שני הביטים מאופסים על ידי מערכת ההפעלה. אחת לזמן מה (לפי פסיקות שעון), R ביט מאופס כדי להבחין בין דפים שלא פנו אליהם לאחרונה, מדפים שכן.
כאשר מתרחשת פסיקת דף, מערכת ההפעלה אוספת את כל הדפים ומחלקת אותן לקבוצות לפי הסיביות M ו- R.
קבוצה 0: ללא פנייה וללא שינוי.
קבוצה 1: ללא פנייה, בוצע שינוי.
קבוצה 2: פנייה ללא שינוי.
קבוצה 3: פנייה עם שינוי.
קבוצה 1 אפשרית כאשר לא פנו לתהליכים שהיו בקבוצה 3 מאז פסיקת השעון האחרונה שבה סיבית R עברה איפוס. פסיקות שעון לא מאפסות את M ביט היות והמידע חשוב כדי לדעת האם להעתיק את הדף לדיסק או לא.
אלגוריתם NRU מוצא מהזיכרון דף באקראי, מהקבוצה הנמוכה ביותר שאינה ריקה. משתמע מהאלגוריתם שעדיף להוציא דף ששונה, אולם לא הייתה פניה אליו, לפחות מפסיקת השעון האחרונה מאשר להוציא דף שיש אליו פניות.
יתרונותיו הם שהוא קל להבנה, ויעיל לביצוע. ביצועיו אמנם אינם אופטימליים, אולם מספקים למדי.

3.4.3 אלגוריתם התור FIFO

כדי להמחיש את אופן פעולת האלגוריתם, נדמה סופרמרקט עם מספר מדפים מספיק כדי להציג K מוצרים שונים. יום אחד, חברה מסוימת מוציאה מוצר חדש שהוא הצלחה מיידית. ולכן על הסופרמרקט שלנו לוותר על מוצר ישן, וזאת על מנת שיוכל להציג את החדש.
אפשרות אחת היא למצוא מוצר שמשווק כבר הרבה זמן, ולהיפטר ממנו, על בסיס הטענה שהקהל לא מעונין בו עוד.
ולכן הסופרמרקט יכול להכין רשימת מוצרים לפי ותק. החדש ביותר בסוף, והישן בהתחלה, כאשר את המוצר הראשון מבטלים.
האלגוריתם פועל באופן דומה. מערכת ההפעלה מחזיקה רשימה של כל הדפים שנמצאים כרגע בזיכרון, כאשר הדף שנמצא בראש הרשימה הוא העתיק ביותר, והדף שבסוף הרשימה הוא הדף שהגיע אחרון. (כלומר ממוין לפי זמן ההגעה לזיכרון).
כאשר מתרחשת פסיקת דף, הדף שבראש הרשימה מוסר והדף החדש מתווסף לסוף הרשימה.

יתרון השיטה – פשטות.
חסרון השיטה הוא שגם דפים שמשתמשים בהם לעיתים קרובות מנופים מהזיכרון. לכן לא משתמשים באלגוריתם זה. בוחנים כאן גיל ולא נחיצות, שבאמת מעניינת אותנו.

3.4.4 אלגוריתם ההזדמנות השניה (FIFO+R BIT)

שנוי קל באלגוריתם התור המונע זריקת דפים הנמצאים בשימוש רב ותכוף מהזיכרון, הוא: כאשר מתרחשת פסיקת דף נבחן R BIT של הדף הותיק ביותר. אם הוא 0 אזי הדף הוא גם ותיק וגם לא בשימוש, והוא מוחלף באופן מיידי. אולם אם סיבית R היא 1, הסיבית מאופסת, הדף עובר לסוף התור, ובזמן ההגעה שלו נרשם שהגיע זה עתה לזיכרון. וכך ממשיך החיפוש. פעולת האלגוריתם נקראת ההזדמנות השניה. ראה איור 24-3.
אם כל הדפים היו שימושיים אזי אלגוריתם זה הופך להיות FIFO טהור. אופן הפעולה הוא שכל הדפים מוזזים אחורה כאשר סיבית R שלהם מאופסת, כאשר הראשון מגיע שוב לתחילת התור, (מהסיבה שכל הדפים היו שימושים) אזי סיבית R שלו מאופסת, והוא מוחלף באופן מיידי, לפיכך האלגוריתם תמיד מסתיים.

עמוד 111

3.4.5 אלגוריתם השעון

למרות שאלגוריתם הזדמנות שניה הוא הגיוני, הוא אינו יעיל היות והוא מעביר כל פעם דפים ברשימה מקושרת. הייעול שמוסיף אלגוריתם השעון הוא באחזקה של רשימה מעגלית שבה אין צורך להזיז דפים. ישנו מצביע שמצביע על הדף הותיק ביותר ברשימה.
כאשר מתרחשת פסיקת דף בוחנים את הדף המוצבע. אם סיבית R היא 0, הדף מנופה והדף החדש נכנס במקומו כאשר מקדמים את המצביע לדף הבא ברשימה. אם סיבית R היא 1, אנו מאפסים את סיבית R והמצביע מקודם לדף הבא אחריו, ובכך דף זה הופך לראש הרשימה, והתהליך חוזר על עצמו עד שנמצא דף שבו סיבית R היא 0.
ההבדל מאלגוריתם ההזדמנות השניה הוא במבנה הנתונים ובכך שמשתמשים כאן במצביע אחד בעוד בהזדמנות השניה יש להחזיק שני מצביעים, ולבצע תזוזות. לכן אלגוריתם השעון מהיר יותר פי שניים.

3.4.6 אלגוריתם הדף הותיק – LRU

הערכה שקרובה לאלגוריתם האופטימלי היא שדפים שהשתמשו בהם הרבה בהוראות האחרונות, כנראה ישתמשו בהם גם בעתיד באופן רב, ולהפך.
על בסיס רעיון זה נבנה האלגוריתם LRU: כאשר מתרחשת פסיקת דף, יש לנפות את הדף שלא השתמשו בו במשך הזמן הארוך ביותר. Least Recently Used.

למרות שתאורטית אלגוריתם זה הגיוני הוא אינו זול בכלל. הקושי ביישום נובע מכך שיש להחזיק רשימה מקושרת עבור כל הדפים בזיכרון כאשר הדף שהשתמשו בו הכי פחות לאחרונה, יהיה בסוף הרשימה. הקושי טמון בכך שיש לעדכן את הרשימה בכל פניה לזיכרון. כמו כן הפעולות הבאות: למצוא את הדף ברשימה, למחוק אותו, להזיז אותו לראש הרשימה, צורכות זמן פעולה רב.

עמוד 112

יש למצוא פתרון חומרה, או למצוא פתרון תכנותי זול יותר.
חיפוש וטיפול ברשימה מקושרת בכל הוראה הוא איטי מאוד, אפילו בחומרה. אולם ישנם דרכים אחרות לביצוע LRU עם חומרה מיוחדת.
פתרון חומרה – יש לצייד את החומרה במונה 64 ביט, שיגדל ב- 1 בכל הוראה. בנוסף לכל דף יהיה שדה גדול מספיק כדי להכיל את ערכו של המונה (של 64 ביט). לאחר כל פניה לזיכרון, ערכו הנוכחי של המונה מאוחסן בכניסה של הדף שפנו אליו עכשיו, בטבלת הדפדוף. כאשר מתרחשת פסיקת דף מערכת ההפעלה בוחנת את כל המונים של הדפים בזיכרון למצוא את הנמוך שבהם והוא זה שמנופה.
פתרון חומרה שני – במחשב עם N מסגרות, החומרה תשמור מטריצה של NXN ביטים, מאותחלת כולה באפסים. כאשר מתקיימת גישה למסגרת K , החומרה שמה בכל השורה

K 1 –ים, ומאפסת אל כל העמודה K. בכל נקודת זמן השורה שיש לה את הערך הבינארי הנמוך מכולם היא זו שהשתמשו בה בזמן הרחוק ביותר. ראה שרטוט 3-26.

אלגוריתם השימוש הנדיר NFU

אמנם אלגוריתם LRU בשני אופניו הוא בר ביצוע, אולם הוא מסתמך על חומרה מיוחדת. לכן נמצא פתרון תכנותי עבור מחשבים החסרים חומרה זו.

עמוד 113

אפשרות אחת נקראת NFU (והוא יישום של LRU). פתרון זה דורש מונה בתוכנה לכל דף מאותחל ל- 0. בכל פסיקת שעון מערכת ההפעלה סורקת את כל הדפים בזיכרון ועבור כל דף ערכה של סיבית R (0 או 1) נוסף למונה.

על ידי כך המונים למעשה משמשים אינדיקציה באיזו תדירות הייתה פניה לכל אחד מהדפים בזיכרון. כאשר מתרחשת פסיקת דף הדף שערך המונה שלו הוא הנמוך ביותר, מנופה.

הבעיה המרכזית עם אלגוריתם NFU היא שהוא "אינו שוכח דבר לעולם". הכוונה היא שכרכי המונה נותרים, ואם יש דף שהשתמשנו בו הרבה אך לאחרונה אנו כלל לא זקוקים לו, אכך המונה לו עדיין ישאר גבוה ולא נזרוק אותו פרק זמן ארוך. לדוגמא במעבד שמבצע כמה מעברים, ניתן להשתמש בתכיפות בדף מסוים במשך מעבר 1, עדיין המונה שלו יהיה גבוה במעברים הבאים. ואם נניח שבמעבר 1 היה זמן הביצוע הארוך מכל המעברים, אזי ערכו יישאר גבוה לתמיד, ומערכת ההפעלה תפנה דף מועיל יותר, במקום הדף שכבר לא בשימוש.

שיפור אלגוריתם NFU -> יישום LRU:

לשיפור יש שני חלקים:

1. הסיביות במונים מוזזים ימינה בסיבית אחת לפני שערכו של R מתווסף אליהם. (על ידי כך המונה קטן אם אין התייחסות).
2. הערך של סיבית R מתווסף לסיבית השמאלית ביותר במקום לסיבית הימנית ביותר (עמוד 113 שרטוט 3-27).

אלגוריתם זה ידוע בשם – AGING(אלגוריתם ההזדקנות). כאשר מתרחשת פסיקת דף הדף שלו הערך הבינרי הקטן מכולם מנופה. ההגיון הוא בכך שדף שלא פנו אליו – יש לו יותר אפסים מצד שמאל. כי כאשר יש פניה נכנס 1 משמאל, אחרת 0.

נניח שאחרי פסיקת השעון הראשונה סיביות R בחמשת העמודים 0 עד 5 יהיו 1,0,1,0,1,1 בהתאמה. במילים אחרות בין פעימה 0 לפעימה 1 הייתה פניה לדפים 0,2,4,5 וסיביות R שלהן נדלקו, בעוד האחרות נותרו 0. אחרי 6 מניות המונים הוזזו, וסיבית R הוכנסה משמאל. התוצאה בשרטוט 3-27 עבור 6 מונים, ב- 4 פעימות שעון.

עמוד 114

ישנם שני הבדלים בין יישום LRU בחומרה בטבלה, לבין יישום LRU בתוכנה.

האלגוריתם LRU בחומרה אינו מבחין בין דפים שלא ניגשו אליהם בין פסיקת שעון אחת לשניה. כלומר LRU בתוכנה מבחין באיזה דף מבין הדפים שלא ניגשו אליהם, ניגשו אחרון.

ביישום LRU בתוכנה יש למונה מספר קבוע של סיביות – 8. כלומר אם ברגע נתון לשני דפים כל הסיביות הן 0 , לא נוכל לדעת איזה שונה אחרון ל- 0, ואיזה שימושי יותר.

3.5 מודלים להחלפת דפים

פרק זה עוסק בניתוח תיאורטי של בעיית החלפת הדפים. מודל הוא סדרת הנחות על תהליך, המאפשרת ניתוח מתמטי של פעולתו. המודל מפשט את המציאות, אולם (כך אנו מקווים), הוא עדיין מייצג נאמנה את התהליך, ולפיכך ניתן להסיק מסקנות אמפיריות מניתוח המודל.

3.5.1 האנומליה של BELADY

באופן אינטואיטיבי, נראה כי ככל שיש יותר מסגרות בזיכרון, יהיו פחות פסיקות דף בתוכנית. באופן מפתיע, הדבר אינו תמיד נכון. BELADY הביא דוגמא נגדית, שבה תור גורם יותר פסיקות דף עם 4 מסגרות בזיכרון, מאשר עם שלוש. מצב מוזר זה נקרא

האנומליה של BELADY. והיא מודגמת באיור 3-28 עבור תוכנית עם 5 דפים ווירטואלים, ממוספרים מ- 0 עד 4. הדפים נקראים לפי הסדר הבא:
0,1,2,3,0,1,4,0,1,2,3,4. באיור אנו רואים כיצד עם שלוש מסגרות, נגרמות 9 פסיקות דף, ועם 4 מסגרות 10 פסיקות דף.

<u>אלגוריתמים המבוססים על מחסנית (STACK ALGORITHEMS)</u>
חוקרים רבים בתחום המחשבים היו המומים בעקבות האנומליה של בלדי, והחלו לחקור בנושא. עבודתם הובילה לפיתוח התיאוריות של אלגוריתמים לדפדוף ותכונותיהם. להלן תאור קצר:
כל המחקר החל עם ההשקפה שכל תהליך מייצר רצף של פניות לזיכרון במהלך ריצתו. כל פניה לזיכרון קשורה לדף וירטואלי מסוים. כך שבאופן רעיוני גישת תהליך לזיכרון יכולה להיות מאופיינת כל ידי רשימה מסודרת של מספרי הדפים. רשימה זו נקראת REFERENCE STRING, ומשמשת תפקיד מרכזי בתיאוריה. למען הפשטות נתייחס למכונה שיש בה רק תהליך אחד ומחרוזת התייחסות אחת. מערכת הדפדוף מאופיינת על ידי שלושה דברים –

1. מחרוזת הפניות של התהליך המבוצע.
2. האלגוריתם להחלפת דף.
3. מספר המסגרות (הדפים הפיזיים) הזמינים בזיכרון, M.

המודל המופשט על פי אלגוריתם המחסנית:
נחזיק מטריצה – M המציגה בכל רגע נתון את מצב הזיכרון. למטריצה יש מספר שורות כמספר הדפים הווירטואלים – N, ומספר עמודות – כאורך רשימת הפניות (REFERENCE STRING).
השורות במטריצה נחלקות לשני חלקים:
החלק העליון M מכיל את הדפים שנמצאים בזיכרון הפיזי.
החלק התחתון N-M מכיל דפים שהיו בזיכרון הפיזי ונופו לדיסק.
כאשר M מאותחל אין כלל דפים בטבלה, היות ועדיין לא הייתה פניה לדף בזיכרון.
כאשר הביצוע מתחיל התהליך מתחיל להוציא דפים מרשימת הפניות, פניה אחת בכל פעם. בכל פניה המפרש (INTERPRETER) בודק האם הדף בזיכרון.

עמוד 116

על ידי מעבר עמודה עמודה מתחילת המטריצה. אם הדף אינו נמצא בחלק העליון (M זיכרון פיזי) אזי מתרחשת פסיקת דף. אם יש מקום פנוי בזיכרון (בתחילת ביצוע התהליך) אזי הדף נטען ומוכנס לחלק העליון של M. אם הזיכרון מלא אזי מעבירים דף שיש לפנות לחלק התחתון והדף החדש מוכנס לחלק העליון. ואת שאר הדפים "דוחפים" למטה. אם הדף נמצא בזיכרון פשוט מעלים אותו לראש העמודה ודוחפים את שאר הדפים למטה.
על מנת להבהיר את פעולת המפרש נתבונן באיור 3-29. הדוגמא משתמשת באלגוריתם LRU לניפוי דפים. למרחב הכתובות הוירטואליות יש 8 דפים, ולזיכרון הפיזי ישנם 4 מסגרות. בחלק העליון של האיור יש לנו רשימת פניות המורכבת מ- 24 דפים:
0,2,1,3,5,4,6,3,7,4,7,3,3,5,5,3,1,1,1,7,2,3,4,1.
מתחת לרשימת הפניות יש 25 עמודות ו- 8 שורות. העמודה הראשונה אשר היא ריקה משקפת את מצב M לפני תחילת הביצוע. כל עמודה משקפת את M לאחר שהדף הגיע מרשימת הפניות על ידי אלגוריתם לדפדוף. הקווים המודגשים מציינים את החלק העליון של M, ואלו ארבעת החריצים הקשורים לארבעת המסגרות בזיכרון הפיזי. דף שעבר מחלק M- 1 זיכרון פיזי, לחלק השני למעשה נופה מהזיכרון הפיזי ועבר לדיסק.
הדף הראשון ברשימת הפניות מספרו 0, והוא נכנס לחלק העליון של הזיכרון, כפי שרואים בעמודה השניה. מספר הדף השני הוא 2, והוא נכנס לחלק העליון של העמודה השלישית. פעולה זו גרמה ל- 0 להידחף כלפי מטה. בדוגמא זו דפים חדשים מושמים למעלה והדפים הקודמים נדחפים כלפי מטה.
כל אחד משבעת העמודים הראשונים ברשימת הפניות גורמים להתרחשות של פסיקת דף. בארבעת הראשונים ניתן לטפל מבלי להוציא דף ישן. הפניה השניה לדף 3 לא גורמת

לפסיקת דף, היות ודף 3 נמצא כבר בזיכרון. למרות זאת המפרש מעביר אותו לראש העמודה. התהליך ממשיך עד לפניה לדף 5.

עמוד 117

דף זה מוזז מהחלק התחתון של M לחלק העליון (למעשה הוא נטען מהדיסק אל הזיכרון). בכל פעם שפונים לדף שלא נמצא בזיכרון הפיזי, מתרחשת פסיקת דף (כפי שמצוין על ידי P בתחתית המטריצה).

נסכם בקצרה את התכונות של מודל זה. כאשר ישנה פניה לדף, הוא תמיד מועבר לכניסה העליונה של M. אם הדף כבר נמצא ב- M הוא מושם בקצה העליון, וכל הדפים מוזזים למטה. מעבר מחלק הראשון של M לשני פירושו מעבר מהזיכרון לדיסק.

אף על פי שדוגמא זו משתמשת באלגוריתם LRU לפינוי דפים, מודל זה פועל גם כן עם אלגוריתמים אחרים, באופן זהה. במיוחד ישנה מחלקה של אלגוריתמים שמעניינים במיוחד : האלגוריתמים שאינם סובלים מבעיית האנומליה של בלדי.

הם נקראים STACK ALGORITHMS, והם בעלי התכונה M(m, r) חלקי או שווה ל- M(m + 1, r), כאשר m הוא משתנה המייצג מסגרת, ו- r הוא אינדקס של רשימת הפניות.

נוסחה זו אומרת שהאלגוריתמים השייכים לקבוצה זו מקיימים מצב שבו אם נגדיל את מספר הדפים הפיזיים באחד ונמשיך את התהליך , שוב, אזי בכל נקודה במהלך הביצוע כל הדפים ששהו מקודם בזיכרון הפיזי יהיו גם כעת בתוספת של דף נוסף בזיכרון הפיזי.

אלגוריתם LRU , ואלגוריתם אופטימלי מקיימים תכונה זו, אלגוריתם FIFO לא.

3.5.3 מחרוזת המרחק

עבור אלגוריתמי המחסנית, נוח להציג את רשימת הפניות בדרך מופשטת יותר מאשר בדרך של מספר דף. התייחסות לדף תהיה לפי המרחק מתחילת העמודה, בפעם האחרונה שהוא מופיע, ומרחק זה ירשם במטריצה בעת הפניה אליו. לדוגמא, הפניה לדף 1 בעמודה האחרונה באיור 3-29 היא פניה לדף במרחק 5 מהקצה העליון של המחסנית (היות ו- 1 היה מספר 5 לפני הפניה).

דפים שעדיין לא פנו אליהם, לפיכך עדיין לא במחסנית (עדיין לא ב- M), נאמר שמרחקם הוא אינסוף.

נציין שמחרוזת המרחק תלויה לא רק ברשימת הפניות, אלה גם באלגוריתם הדפדוף. עם אותה רשימת פניות, אלגוריתם דפדוף אחר, יבצע בחירות אחרות, לגבי הדפים שיש לנפות. כתוצאה מכך יהיה רצף אחר במחסנית.

לתכונות הסטטיסטיות של מחרוזת המרחק יש השפעה גדולה על ביצועי האלגוריתם.

האיור 3-30 הם היסטוגרמה של המספרים המופיעים במחרוזת המרחק. באיור הראשון יש הצטברות גדולה של המספרים מ- 1 ועד K, לפיכך זיכרון מדומה בעל K מסגרות לפחות יקטין במידה משמעותית את מספר שגיאות הדף לגבי האלגוריתם שעבורו חושבה היסטוגרמה זו. באיור השני, הפיזור האחיד מראה כי כל הקטנה של מספר המסגרות מעבר למספר המקסימלי תקטין במידה רבה את יעילות האלגוריתם.

מחרוזת המרחק היא איפוא כלי חישובי לקביעת מספר המסגרות הדרושות למימוש יעיל של אלגוריתם מחסנית נתון.

עמוד 118

3.5.4 חיזוי מספר שגיאות הדף

אחת מהתכונות הטובות של מחרוזת המרחק היא שניתן לחזות באמצעותה את מספר שגיאות (פסיקות) הדפים המתרחשות בזיכרון בעל גדלים שונים. האלגוריתם מתחיל על ידי סריקת מחרוזת המרחק, דף אחר דף. היא שומרת את מספר הפעמים ש- 1 מופיע, מספר הפעמים ש- 2 מופיע וכך הלאה. יהי Ci מספר הפעמים שדף I מופיע. עבור מחרוזת המרחק מאיור 3-29 מערך C משורטט באיור 3-31. בדוגמא זו קורא 4 פעמים שפונים לדף שכבר נמצא בראש המחסנית, 3 פעמים לדף שאחריו וכן הלאה. C עם הסימן אינסוף יהיה מספר הפעמים שאינסוף מופיע במחרוזת המרחק.

נוסחה המאפשרת חיזוי מספר המסגרות האופטימלי :

Fm שווה לסיגמא מ-k=m+1 ועד n של Ck + C& (הסימן & משמש לסימון אינסוף).

ערכו של Fm הוא מספר שגיאות הדף שיקרו, במרחק נתון במחרוזת המרחק עם m מסגרות בזיכרון. עבור מחרוזת המרחק באיור 3-29, איור 3-1 ניתן מערך F. לדוגמא, F1 הוא 20, הכוונה היא שהזיכרון מחזיק רק מסגרת אחת, ומתוך 24 פניות במחרוזת, בכולם תתרחש שגיאת דף, למעט הארבעה הזהים לפניה הקודמת לדף.
כדי להבין מדוע נוסחה זו פועלת נלך לאיור 3-29, לריבוע המודגש (מספר המסגרות בזיכרון הפיזי). יהי m מספר מסגרות הדפים בחלק העליון של M. שגיאת דף מתרחשת בכל פעם שפניה ממחרוזת המרחק היא m+1 או יותר.
הסיגמה בנוסחה היא עבור חיבור מספר הפעמים שדבר זה מתרחש. מודל זה יכול לשמש עבור חיזויים נוספים גם כן.
לפי נוסחה לחישוב F ברור כי תמיד מתקיים Fm <= F(m+1).
ההנחות הנדרשות לחיזוי זה הן:

א. מספר הדפים המקסימלי.
ב. מחרוזת המרחק.
מחרוזת המרחק תלויה בשני גורמים:
1. סדרת ההתייחסויות (פניות).
2. אלגוריתם הדפדוף. אלגוריתם זה חייב להיות אלגוריתם מחסנית!
ג. מספר המסגרות המקסימלי האפשרי.

עמוד 119

3.6 שיקולים בתכנון של מערכות דפדוף בזיכרון

בסעיף זה אנו עוברים משיקולים תיאורטיים לשיקולים הנדסיים של בניית המערכת עצמה. כאשר מנסים להפעיל מערכת דפדוף הבנויה על פי העקרונות שתיארנו עד כה, מתגלות תופעות מעניינות, המאפשרות ייעול נוסף בתכנון מערכת הדפדוף.

3.6.1 מודל קבוצת העבודה

במצב ההתחלתי, כאשר התהליך מתחיל לרוץ, אין כלל דפים בזיכרון. כאשר המעבד מבצע את ההוראה הראשונה מתרחשת פסיקת דף, הגורמת למערכת ההפעלה להביא את הדף הדרוש להוראה הראשונה. לאחר מכן במהירות יחסית יתרחשו עוד מצבי שגיאות דף עבור משתנים גלובליים, ופעילות מחסנית. וכך לאחר זמן מה רוב הדפים הדרושים לתהליך יהיו בזיכרון והתהליך יוכל לרוץ עם מספר קטן יחסית, של שגיאות דף.
האסטרטגיה שתוארה לעיל קרויה בשם DEMAND PAGING היות והדפים נטענים לזיכרון בעקבות דרישה ולא נטענים מראש.
כמובן, קל לכתוב תוכנית לניסיון, שבאופן עקבי קוראת את כל הדפים שבמרחב הכתובות, ושתגרום לכל כך הרבה פסיקות דף שלא יהיה מספיק זיכרון להכיל את כולן. למרבה המזל רוב התהליכים אינם פועלים בצורה זו.
המונח LOCALITY OF REFERENCE, הנחת המקומיות, פירושו שבמהלך כל אחד משלבי הביצוע התהליך פונה לחלק קטן מהדפים. קבוצת הדפים שתהליך משתמש בהם נקראת קבוצת עבודה WORKING SET.

עמוד 120

זהו אולי המודל החשוב ביותר לתכנון המערכת. ההנחה העומדת ביסודו – הנחת המקומיות, היא ההנחה העומדת במובלע ביסוד רובם של האלגוריתמים לדפדוף.
אלגוריתם קבוצת העבודה מנסה לאמוד את גודלה של קבוצת הדפים שהתהליך מתייחס אליה. ההנחה היא כי שימור קבוצת דפים זו בזיכרון כל עוד התהליך פעיל, תקטין למינימום את מספר שגיאות הדף הקשורות לתהליך זה.
אם כל קבוצת העבודה נמצאת בזיכרון, תהליך יפעל מבלי לגרום לשגיאות דף רבות, עד שיעבור לחלק ביצועי אחר. אם הזיכרון הפנוי קטן מדי מלהכיל את כל קבוצת העבודה, התהליך יגרום להרבה שגיאות דף וירוץ לאט מאוד.
כאשר תהליך שיוצר פסיקות דף כה מרובות עד שכמעט אינו יכול להתקדם, אנו אומרים כי התהליך נכנס למצב של סחרור (THRASHING). מודל קבוצת העבודה אמור לצמצם תופעה זו ככל האפשר.

במערכת TIMESHARING, תהליכים מועברים לעיתים קרובות לדיסק (הדפים שלהם מועברים מהזיכרון), כדי לאפשר לתהליכים אחרים לקבל זמן מעבד.
השאלה היא מה לעשות כאשר מחזירים את התהליך לזיכרון שוב. טכנית, לא צריך לעשות דבר. התהליך יגרום לפסיקות דף עד שתטען קבוצת העבודה שלו. הבעיה היא שאם יש בין 20 ל- 100 פסיקות דף, בכל פעם שתהליך נטען, פעולתו תואט, וזה גם מבזבז את זמן המעבד, היות ולוקח למערכת ההפעלה כמה מילי-שניות של זמן מעבד לעבד פסיקת דף. לכן הרבה מערכות דפדוף עוקבות אחרי כל קבוצת עבודה של כל תהליך, כדי להבטיח שקבוצת העבודה תטען לזיכרון לפני שהתהליך יתחיל לרוץ. גישה זו נקראת WORKING SET MODEL, מודל קבוצת העבודה. והוא תוכנן על מנת להקטין את אחוז פסיקות הדף. טעינת הדפים לפני שתהליך רץ נקראת PREPAGING.
בכדי ליישם את מודל קבוצת העבודה יש למצוא דרך לדעת איזה דפים שייכים לקבוצת העבודה. לשם כך ניתן להסתייע באלגוריתם AGING שתואר. כל דף המכיל 1 בין N הביטים הגבוהים במונה, נחשב לחבר בקבוצת העבודה. עם לא הייתה פניה לדף במשך N פעימות שעון רצופות, הוא יוצא מקבוצת העבודה. הפרמטר N נקבע אחרת בכל מערכת, אולם ביצועי המערכת בדרך כלל לא רגישים לערך המדויק של N.
מידע על קבוצת העבודה יכול לשמש לשיפור הביצוע של אלגוריתם השעון. בדרך כלל אם סיבית R היא 0 אזי מנפים את הדף, השיפור מתבטא בכך שבודקים האם הדף הוא חלק מקבוצת העבודה של התהליך הנוכחי, אם כן אזי לא מנפים אותו. אלגוריתם זה קרוי WSCLOCK.

<u>3.6.2 מדיניות החלפה מקומית לעומת מדיניות החלפה גלובלית</u>
עד עתה התעלמנו בדרך כלל מן העובדה שעל הזיכרון מתחרים תהליכים רבים. שאלה חשובה היא – איך הזיכרון מוקצה בין התהליכים המתחרים?
איור 3-32 מראה 3 תהליכים A,B,C רצים. נניח ו- A מקבל פסיקת דף.
<u>עמוד 121</u>
האם צריך אלגוריתם ניפוי דפים למצוא את הדף האחרון שהיה בשימוש, תוך שמכלילים רק את ששת הדפים שהוקצו לאחרונה ל- A, או להכליל את כל הדפים שבזיכרון? האם להכליל באלגוריתם ניפוי דפים את כל התהליכים או רק את התהליך שקרה לו מצב של פסיקת דף.
אם נתחשב רק ב- A, הדף בעל הדרוג הנמוך ביותר הוא A5, ואז נוצר המצב באיור 3-32b. מאידך, אם ננפה את הדף בעל הדרוג הנמוך ביותר ללא קשר לאיזה תהליך הוא שייך, אזי דף B3 יבחר לניפוי. האלגוריתם שמנפה רק מהתהליך שגרם לפסיקת דף נקרא LOCAL REPLACEMENT ALGORITHM(מדיניות מקומית) ואילו האלגוריתם אשר בעת פסיקת דף מכליל את כל הדפים שבזיכרון השייכים לכל התהליכים נקרא GLOBAL ALGORITHMS (מדיניות גלובלית).
מדיניות מקומית – מקצה לכל תהליך שטח קבוע בזיכרון וכאשר קורה מצב של פסיקת דף, מכלילים באלגוריתם הניפוי רק את אותו תהליך.
מדיניות גלובלית – מקצים את הזיכרון באופן דינמי בין כל התהליכים שרצים בזיכרון, ובמצב של פסיקת דף מכלילים באלגוריתם הניפוי את כל התהליכים בזיכרון.
באופן כללי אלגוריתמים שעובדים לפי המדיניות הגלובלית פועלים טוב יותר, בייחוד כאשר קבוצת העבודה יכולה להשתנות במהלך חייו של התהליך. אם משתמשים באלגוריתם גלובלי, המערכת צריכה להחליט באופן מתמיד כמה יחידות זיכרון (מסגרות דפים) להקצות לכל תהליך, על מנת שמצד אחד כל קבוצת העבודה תהיה בזיכרון למניעת סחרור, ומצד שני לא להקצות דפים מעבר לגודלה של קבוצת העבודה למניעת בזבוז.
חשוב לזכור כי גודל קבוצת העבודה משתנה כל הזמן במשך חיי תהליך.
גישה אחת מציעה אלגוריתם שמקצה מסגרות לתהליכים. דרך אחת היא להחליט לקבוע את מספר התהליכים הרצים בכל פרק זמן ולהקצות לכל תהליך חלק שווה. לפיכך עם 475 מסגרות פנויות ועשרה תהליכים, כל תהליך יקבל 47 מסגרות. חמשת הנותרים ייוותרו לשימוש כאשר תקרה פסיקת דף.

אמנם גישה זו נראית הוגנת, אין הגיון רב בנתינת חלק שווה של הזיכרון לתהליך בגודל 10K ולתהליך בגודל 300K.

עמוד 122

במקום, ניתן להקצות דף באופן פרופורציוני לגודל כל תהליך. כמו כן יהיה נבון לקבוע מינימום הקצאה מסוים, בלא להתחשב בגודל.
גם ההקצאה הקבועה, וגם ההקצאה הפרופורציונית לא מתמודדות עם בעיית הסחרור.
גישה המטפלת בכך היא PAGE FAULT FREQUENCY או אלגוריתם הקצאה PFF.
עבור קבוצה גדולה של אלגוריתמים לניפוי דפים כולל LRU ידוע שאחוז פסיקות הדף יורד ככל שמוקצים יותר דפים. באיור 3-33 דיאגרמה עבור אחוז פסיקות הדף כפונקציה של מספר המסגרות המוקצות בזיכרון.
הקו המקווקו A הוא אחוז שגיאות הדף גבוה מידי, כך שלתהליך הגורם לכך ינתנו יותר מסגרות על מנת להפחית את אחוז השגיאות. הקו המקווקו B מצביע על אחוז כה נמוך של שגיאות דף, שניתן להסיק מכך שלתהליך הוקצה יותר מידי מקום בזיכרון. במקרה זה יתכן ויילקחו ממנו מסגרות. אם יתגלה שיש בזיכרון הרבה תהליכים שלא ניתן לשמור אותם מתחת ל- A, יועברו כמה מהתהליכים מהזיכרון, והמסגרות שלהם יחולקו בין התהליכים הנותרים, או יהוו מקור לדפים פנויים שישתמשו בהם בעת פסיקות דף.
ההחלטה לפנות תהליך מהזיכרון, היא צורה של ניהול טעינת תהליכים. אנו רואים שאפילו עם דפדוף, עדיין זקוקים להחלפה, כדי להפחית דרישה פוטנציאלית מהזיכרון.
המנגנון המוצע הוא סטטיסטי: מודדים את תדירות פסיקות הדף של כל תהליך ובמידת הצורך מוסיפים לו דפים. ברור כי המחיר מתבטא לעיתים קרובות בהקטנת מספר התהליכים בזיכרון. באותה צורה ניתן לבדוק עד כמה תשפיע הקטנת קבוצת העבודה של תהליך על תדירות שגיאות הדף שלו. אם השפעה זו אינה ניכרת זהו סימן לכך שקבוצת העבודה התכווצה.
מדיניות לוקלית – אם משתמשים במדיניות לוקלית וקבוצת העבודה גדלה, הדבר יביא לסחרור. למרות שייתכן שיש מספיק דפים פנויים בזיכרון (בגלל שהתהליכים האחרים קטנים יותר ברגע נתון) לעומת זאת אם קבוצת העבודה מצטמקת, המדיניות הלוקלית תביא לבזבוז זיכרון.

3.6.3 שיקולים לקביעת גודלו של דף

גודל הדף הוא לעיתים קרובות פרמטר הניתן לקביעה על ידי מתכנני מערכת ההפעלה.
קביעת גודל אופטימלי של דף דורש שקלול של כמה דרישות מתחרות.

עמוד 123

שיקולים ליצירת דף קטן:
במקרה הממוצע, מחצית הדף האחרון (בסדרה של דפים המוקצים) נותרת ריקה ולכן מבוזבזת.
בעזרת דפים קטנים ניתן לדייק יותר בהתאמת מספר הדפים בזיכרון עבור תהליך מסוים, לגודלה של קבוצת העבודה שלו. כלומר זה ימנע מצבים בהם תוכניות שלא משתמשים בהן יהיו בזיכרון.
מצד שני, השיקולים ליצירת דף גדול:

1. המשמעות של דפים קטנים היא שתוכניות יזדקקו להרבה דפים, לכן טבלת הדפים תגדל. לכן דפים גדולים מקטינים את גודלה של טבלת הדפים.
2. מהירות העברת דפים אל הדיסק וממנו אינה מושפעת במיוחד מגודלם. דפים גדולים מבטיחים אפוא מהירות העברה גדולה יותר.

מחשבים מסוימים טוענים את טבלת הדפדוף לרגיסטרים בכל פעם שהמעבד מחליף בין תהליך אחד למשנהו. במחשבים אלו דפים קטנים פירושם שהזמן שיידרש לטעינת הרגיסטרים יתארך ככל שהדפים יקטנו. ניתן לסכם נקודה זו בצורה מתמטית. יהי S גודל התהליך הממוצע, וגודל הדף P. נניח שכניסת דף דורשת E ביטים. המספר המשוער של דפים שהתהליך זקוק להם הוא S\P , הדורשים SE\P ביטים בטבלת הדפדוף. הזיכרון המבוזבז בדף האחרון הוא P\2. לפיכך התקורה לטבלת דפדוף, ובזבוז הזיכרון הוא OFERHEAD=SE\P+P\2. החלק הראשון (גודל טבלת הדפדוף) גדל ככל שגודל הדף

קטן. החלק השני (חלוקה פנימית) גדל כאשר הדף גדל. האופטימום נמצא אי שם באמצע. ולאחר פיתוח נקבל שגודל הדף P שווה לשורש של 2SE.

עמוד 124

3.6.4 שיקולי יישום

כאן אנו עוברים ליישום הרעיונות הקודמים, ונתקלים בבעיות הקשורות למבנה המחשבים עצמם. בהמשך נעמוד על כמה מהן.

גיבוי הוראות מכונה

כאשר מתרחש מצב של פסיקת דף, ולא ניתן להשלים הוראה בשל כך, יש צורך "לגלגל" את הפעולה לאחור (חזרה מדויקת למצב שלפני ההוראה שנפסקה) ולהתחיל את הפעולה מחדש לאחר שהדף הבא כבור הובא לזיכרון. כדי לגלגל פעולה לאחור יש למעשה צורך במנגנון של גיבוי ההוראה האחרונה על מנת לאפשר לזכור את מצב המעבד וסביבתו (מונה התוכנית, מצב האוגרים וכו'), כפי שהיו לפני תחילת הפעולה. רצוי מאוד שגיבוי זה יעשה בחומרה. במחשבים מסוימים ישנו רגיסטר אשר לפני ביצוע כל הוראה, מועתק אליו מונה התוכנית. כאשר אין גיבוי אוטומטי בחומרה, מתכנני מערכות הפעלה נדרשים למצוא פתרונות (מסובכים הרבה יותר) בתוכנה.

עמוד 125

נעילת דפים בזיכרון

הצורך בנעילה (כלומר הפסקה של אפשרות הדפדוף לגבי דף מסוים ובעקבות זאת מניעת הוצאתו מהזיכרון) מתעורר כאשר התקן קלט\פלט (למשל הדיסק) מעביר או קורא נתונים מהזיכרון (ממשתנה מאגר כלשהו בזיכרון המכיל את הנתונים). העברה זו קרויה העברה ישירה – DMA והיא תפסק ותשתבש אם במהלכה יוחלף הדף בו נמצא משתנה המאגר. חלק מהנתונים יהיו במאגר וחלקם יכתבו לתוך דף חדש שהוכנס לזיכרון. פתרון אפשרי אחד הוא נעילת הדף בזיכרון עד לגמר ההעברה. פתרון אחר הוא להעביר את הנתונים לדפים שבשליטת מערכת ההפעלה (KERNEL MODE) ואחר כך להעתיק את הנתונים לדפי משתמש.

בפתרון זה הדפים של ה- KERNEL אינם מוחלפים וכמו-כן כאשר מועברים הנתונים מדפי ה- KERNEL ל- USER הדבר נעשה בשליטה ופיקוח של מערכת ההפעלה על הרגע שבו יוחלף התהליך. ומכיוון שמתבצעת פעולת העתקה בבת אחת הדף נמצא בביקוש ולכן אין סכנה שיוחלף.

דפים לשימוש משותף

קיימת אפשרות להשתמש בו זמנית במספר דפים עבור קבוצת תהליכים (מספר משתמשים המריצים תהליכים במקביל, למשל EDITOR או קומפיילר). כך יכולות תוכניות הנמצאות בשימוש רב, כמו מעבדי תמלילים, להופיע בזיכרון רק פעם אחת, בעת שתהליכים רבים משתמשים בהן.

עמוד 126

קיימת סכנה שפעולת כתיבה של תהליך אחד תשפיע על מצבם של תהליכים אחרים. הפתרון לבעיה זו הוא לאפשר שיתוף בדפים שהם לקריאה בלבד.

בעיה נוספת קשורה לעובדה שדפים משותפים אינם יכולים להתפנות מהזיכרון כאשר אחד התהליכים שהשתמש הם מסתיים. ניתן לפנות דפים אלו רק כאשר התהליך האחרון גמר את השימוש בדפים המשותפים. לעתים, דרישה זו מנוגדת למדיניות הנקבעת על ידי אלגוריתם החלפת הדפים ודורשת גם מנגנון מיוחד למעקב אחרי השימוש בדפים המשותפים. (מכיוון שמעבר על טבלאות כל התהליכים לגלות דפים משותפים גוזל זמן רב). כמו במקרים רבים אחרים, השאיפה לחסוך זיכרון באמצעות שיתוף דפים יוצרת סיבוך נוסף במערכת הדפדוף כולה.

אחסון בדיסק

בדיון שערכנו בנושא אלגוריתמים להחלפת דפים ראינו כיצד נבחר דף לפינוי אולם לא פרטנו את נושא האחסנה בדיסק של הדף שפונה. נתאר כמה מהנושאים הקשורים בניהול הדיסק.

האלגוריתם הפשוט ביותר להקצאת מקום עבור דף בדיסק הוא להקצות מקום מיוחד להחלפה על הדיסק. כאשר המערכת עוצרת, אזור זה ריק, והוא מיוצג בזיכרון ככניסה יחידה עם מיקומו וגודלו. כאשר מתחיל תהליך לפעול, חלק מאזור זה, בגודל של התהליך,

מוקצה לתהליך, וסך המקום המיוחד להחלפה מפחית את הגודל הזה (של התהליך שרץ).
בכל פעם שתהליך חדש נוסף מוקצה לו מקום לפי גודלו. כאשר הם מסיימים, החלק שלהם בדיסק מתפנה. אזור ההחלפה בדיסק מנוהל כרשימה של חלקים פנויים. הכתובת של אזור אחסון התהליך בדיסק נרשם בטבלת התהליך. חישוב הכתובת לכתיבת דף פשוטה מאוד – יש להוסיף את ההיסט של הדף במרחב הכתובות הוירטואליות עם תחילת אזור ההחלפה. אולם, לפני שתהליך יכול לרוץ, יש לאתחל את אזור ההחלפה. דרך אחת היא להעתיק את כל התהליך לאזור זה ולהביא אותו משם לזיכרון, והשניה היא לטעון את כל התהליך לזיכרון ולהעביר חלקים ממנו לדיסק. בכל אופן במודל פשוט זה ישנה בעיה: גודל תהליכים יכול לגדול, אחרי הריצה. אמנם טקסט התוכנית הוא קבוע, אולם אזור הנתונים יכול לגדול לפעמים, והמחסנית יכולה לגדול תמיד. כתוצאה מכך אולי עדיף להקצות מקום נפרד באזור ההחלפה עבור הטקסט, הנתונים והמחסנית, ולתת להם יותר מקום.

דרך נוספת היא לא להקצות מראש, אלא להקצות מקום בדיסק לכל דף, כאשר הוא מפונה, ולפנותו כאשר הוא נקרא לזיכרון. בדרך זו תהליכים בזיכרון לא קשורים לאזור מסוים בדיסק. החיסרון הוא שיש לאחסן את כתובת הדף בדיסק, בזיכרון כדי לדעת כיצד למצוא כל דף.

האחסון בדיסק יכול להיות מחולק על פי תהליכים או על פי דפים.

באחסון על פי תהליכים: אין צורך לשמור את כתובתו של כל דף.

באחסון על פי דפים: אין צורך לבנות מנגנון לטיפול בגידולו של תהליך תוך כדי פעולתו.

לפתרון של הבאת כל התהליך לזיכרון ברגע טעינתו יש יתרונות מעניינים:

1. הבאת קבוצה גדולה של דפים היא זולה.
2. הדפים המיותרים מתפנים במהירות וללא עלות, שכן אלו דפים נקיים (טרם נכתב בהם דבר חדש).
3. בצורה זו מתגבשת בקלות קבוצת העבודה הראשונה.

החיסרון העיקרי הוא שהדפים המיותרים עלולים לתפוס מקום של דפים החיוניים לתהליך אחר.

<u>שדים מדפדפים</u>

זהו שד המייעל את מערכת הדפדוף. כמרבית השדים במערכת, רוב חייו עוברים עליו בשינה. מידי זמן קצוב הוא מתעורר ובודק את מצב המסגרות הפנויות. אם יש צורך, הוא מוסיף מסגרות למאגר המסגרות הפנויות, וזאת – ללא פינוי המסגרות עצמן, כך שאם דף חדש נדרש מחדש, אין צורך להביא אותו שוב.

הדפדוף פועל במיטבו כאשר יש הרבה מסגרות פנויות, שניתן לבקש כאשר מתרחשת פסיקת דף. אם כל המסגרות מלאות, או מעבר לכך, עברו שינויים אזי לפני שדף חדש יוכל להיכנס, יש לכתוב את הדף הישן לדיסק. כדי להבטיח אספקה של מסגרות משתמשות מערכות דפדוף רבות ברקע בתהליך הנקרא PAGING SAEMON, שד הדפדוף, אשר רוב הזמן ישן, אולם מתעורר לפרקים כדי לבדוק את מצב הזיכרון. אם יש יותר מידי מסגרות פנויות, אזי מתחיל שד הדפדוף לבחור דפים לפינוי, תוך שימוש באלגוריתם החלפה.

<u>עמוד 127</u>

אם הדף עבר שינוי מאז טעינתו לזיכרון, הוא כותב אותו לדיסק.

בכל מקרה, התוכן הקודם של הדף נשמר. במקרה שזקוקים לאחד מהדפים המפונים, לפני שהמסגרת שלו שוכתבה, ניתן לדרוש אותו חזרה על ידי הבאתו ממאגר המסגרות הפנויות. אחזקת אספקה של מסגרות מיעלת את הביצועים מאשר ניצול הזיכרון כולו, ורק שצריך לנסות למצוא מסגרת . לכל הפחות, מבטיח שד הדפדוף שכל המסגרות נקיות, ולא יוצר צורך לכתוב אותם לדיסק במהירות ותחת לחץ כאשר הם ידרשו.

<u>עמוד 128</u>

<u>3.7 חלוקת הזיכרון לקטעים</u>

כד כה התייחסנו אל הזיכרון כאל מיקשה אחת – רצף של כתובות עוקבות. עבור בעיות רבות נעדיף לראות את הזיכרון כמערך של קטעי זיכרון בלתי תלויים זה בזה. כל קטע זיכרון קרוי SEGMENT.

אחד היתרונות המרכזיים של חלוקת הזיכרון לקטעים הוא פתרון אוטומטי לבעיית השטחים המשתנים בגודלם (כגון נתונים או מחסנית) כמודגם בהבדל בין שרטוט 3-35 לשרטוט 3-36.

עמוד 129

כל סגמנט מורכב מרצף לינארי של כתובות, מ- 0 למקסימום מסוים. לסגמנטים שונים, אורכים שונים. אורכו של סגמנט יכול להשתנות במהלך ביצוע תהליך. אורכו של סגמנט המחסנית יכול לגדול בכל פעם שנדחף נתון למחסנית, ולקטון כאשר יוצא נתון מהמחסנית.

היות וכל סגמנט מורכב ממרחב כתובות שונה, סגמנטים שונים יכולים לשנות את גודלם באופן עצמאי, מבלי להשפיע על אחרים. אמנם סגמנטים יכולים להתמלא, אולם בדרך כלל הם מאוד גדולים, כך שאפשרות זו נדירה. כדי לציין כתובת בתוך הסגמנט, התוכנית צריך לתת כתובת המורכבת משני חלקים: מספר סגמנט, והכתובת בתוך הסגמנט. איור 3-36 מדגים זיכרון הבנוי מסגמנטים הנמצאים בשימוש עבור טבלת המהדר.

נדגיש שהסגמנט הוא ישות מקומית, אשר המתכנת מודע לה, ומשתמש בה כישות לוגית מסוימת. הסגמנט יכול להכיל פרוצדורה, או מערך, או מחסנית, או אוסף של משתנים, ובדרך כלל אינו מכיל ערוב של טיפוסים שונים.

יתרונות נוספים בסגמנטציה מחוץ לפשטות בטיפול במבני הנתונים שמשנים את גודלם. אם כל פרוצדורה תופסת סגמנט נפרד, בכתובת 0 ככתובת תחילית, ביצוע הקישור של הפרוצדורות באופן נפרד הוא פשוט.

עמוד 130

כלומר ניתן לתקן פרוצדורה מסוימת מבלי צורך לשנות את שאר הקטעים בתוכנית. כאשר הזיכרון הוא רצף עוקב של כתובות, חלקי התוכנית תלויים יותר זה בזה.

הסגמנטציה מקלה על שיתוף בפרוצדורות או נתונים בין כמה תהליכים. קיימת אפשרות לשימוש בספריות משותפות. לאפשרות זו השלכות עצומות על פיתוח התוכנה ועל גודלו של הקוד המתקבל. דוגמא נפוצה היא ספריה משותפת. בתחנות עבודה מודרניות, המריצות מערכות WINDOW מתקדמות, לעיתים קרובות מכילות ספריה גרפית גדולה, שכמעט כל תוכנית משתמשת בה. במערכת סגמנטים, ניתן לשים את הספרייה בסגמנט, כך שתהליכים מקבילים יוכלו להתחלק בה, תוך ביטול הצורך שהספרייה תיכלל במרחב הכתובות של כל אחד מהתהליכים. ניתן לחלוק בספריות במערכות דפדוף, אולם הדבר יותר מסובך. למעשה מערכות אלו עושות זאת על ידי חיקוי של סגמנטציה.

היות וכל סגמנט מיצג ישות לוגית, אשר המתכנת מודע לה כמו פרוצדורה, מערך או מחסנית, לסגמנטים שונים יש סוגים שונים של הגנות. ניתן לציין סגמנט של פרוצדורה, בהרשאת ביצוע בלבד תוך מניעת ניסיונות לקרוא ממנה, או לאחסן בה. מערך ניתן לציין כהרשאת קריאה\כתיבה, אולם לא ביצוע. הגנות אלו עוזרות לאיתור שגיאות המתכנת.

עמוד 131

ננסה להבין מהו ההגיון העומד מאחורי ההגנות בסגמנטים, אולם לא בזיכרון חד ממדי של דפדוף. בזיכרון סגמנטים המשתמש מודע לתכולת כל סגמנט. בדרך כלל, סגמנט לא יכיל פרוצדורה ומחסנית, לדוגמא, אלא אחד מהשניים. היות וכל סגמנט מכיל טיפוס אחד, ניתן להתאים את ההגנה לטיפוס שבסגמנט. השוואה בין דפדוף לסגמנטציה מוצגת באיור 3-37.

לעומת זאת, תכולת הדף היא מקרית. המתכנת לא מודע לדפדוף כלל וכלל. כך שאם נשים כמה ביטים בכניסה לטבלת הדפים לציין את ההרשאה המותרת, אזי יצטרך המתכנת לעקוב אחר מרחב הכתובות שבו נמצא הדף, וזה בדיוק סוג האדמיניסטרציה שדפדוף הומצא כדי למנוע. היות ולמשתמש בזיכרון סגמנטים ישנה אשליה שכל הסגמנטים נמצאים בזיכרון הראשי כל הזמן, הוא יכול לפנות אליהם כאילו היו, והוא יכול להגן על כל סגמנט בנפרד, מבלי לדאוג לניהול בפועל.

החיסרון היחיד המופיע בטבלה 3-37 הוא הצורך במודעות המתכנת לחלוקת הזיכרון לקטעים. זהו חיסרון לא פשוט, שכן אין ספק שככל שהמתכנת חייב לדעת פחות, כך גדלים סיכוייו להימנע משגיאות.

3.7.1 יישום חלוקת הזיכרון לקטעים

היישום של סגמנטציה שונה מדפדוף בהיבט אחד מרכזי, דפים הם קבועים בגודלם בעוד שסגמנטים לא. איור 3-38 מראה דוגמא של זיכרון פיזי המאותחל בחמישה סגמנטים. מה יקרה אם סגמנט 1 יפונה, וסגמנט 7, שהוא קטן יותר יושם במקומו.

עמוד 132

אזי יווצר בין סגמנט 7 לסגמנט 2 שטח בלתי מנוצל, חור.

אזי נחליף סגמנט 4 בסגמנט 5, ואת סגמנט 3 נחליף בסגמנט 6 (ראה איור 3-38). אחרי שהמערכת תרוץ במשך זמן מה הזיכרון יחולק לקטעים, חלקם מכילים סגמנטים, וחלקם מכילים חורים, דבר המבזבז זיכרון.
במילים אחרות הבעיה המרכזית היא: בעיית שטחים מבוזבזים. כאשר במקום בו שהה סגמנט גדול במקומו נכנס סגמנט קטן, מקבלים שטח מבוזבז.
פתרון לבעיה: כיווץ כל הסגמנטים כלפי מעלה. 3-38e.
מחיר הפתרון: זמן העברת השטחים. העברת כמויות גדולת של נתונים בזיכרון היא פעולה יקרה.
לאחר דחיסת השטח מתעוררת בעיה חדשה כאשר אחד מן הקטעים מתחיל לגדול.

3.7.2 חלוקה לקטעים במערכת MULTICS

אם הסגמנטים גדולים, יהיה זה לא נוח ואפילו בלתי אפשרי לשמור אותם בזיכרון הראשי במלואם. ולכן נולד הרעיון של דפדוף סגמנטים. כך שרק הדפים שבאמת דרושים יהיו בזיכרון. נביא מערכת זו כדוגמה למערכת בה משתמשים בדפדוף של סגמנטים (מכיוון שהם גדולים מדי ואינם יכולים להיות מאוחסנים בזיכרון בבת אחת).
מאפייני המערכת:

א. כל תהליך יכול להכיל עד כרבע מיליון קטעים, כל אחד בגודל 64K.

ב. כל קטע מדופדף לחוד.

ג. לכל תהליך יש טבלת קטעים המהווה את החד הקטעים, ולפיכך גם היא עצמה מדופדפת. טבלה זו מכילה מצביע למיקומו של הדף (בזיכרון הראשי או בדיסק) וכן מידע על גודלם האמיתי של הקטעים (לא תמיד התהליך מנצל את כל ה- 64K העומדים לרשותו), מידע על אפשרויות השיתוף של הקטע ומידע המאפשר את ההגנה עליו.

ד. מבנה הכתובות הוא אופייני לזיכרון מקוטע: כל כתובת מתחלקת לשני חלקים: כתובת הקטע והכתובת הפנימית בתוך הקטע עצמו. הכתובת הפנימית מחולקת למספר הדף ולמילה בתוכו.
כיצד מאותרת כתובת?

א. מפרידים מהכתובת את כתובת הקטע.

ב. המערכת בודקת אם הקטע כבר נמצא בזיכרון. אם כן, האיתור נמשך. אחרת, מופיעה פסיקת דף, והתהליך יוצא למנוחה עד שהקטע יגיע לזיכרון. מקרה מיוחד הוא כאשר הפנייה איננה חוקית, למשל, פלישה לאזור אסור. במקרה זה מועברת הודעה על כך למנהל הזיכרון.

ג. לאחר שהקטע נמצא, יש למצוא את הדף המתאים על פי טבלת הדפים. כרגיל, אם הדף איננו נמצא בזיכרון הראשי, הוא מובא לשם לאחר פסיקת דף.

ד. בשלב זה מאותרת הכתובת בתוך הדף עצמו.

ה. עתה הכל מוכן לקריאה או כתיבה.

מהו היתרון של דפדוף מקומי בכל קטע לחוד?
סביר להניח שדפדוף בכל קטע בפני עצמו יהיה יעיל בגלל עקרון המקומיות. לרוב, המקומיות והמודולריות הולכות יד ביד.
תהליך ארוך יחסית זה עלול להאט ביותר את פעולת המחשב, ולכן קיימת תמיכה בחומרה לסריקה ולחיפוש מהיר במאגר כתובות.

עמוד 145
פרק 4 : מערכות קבצים

א. יש לאפשר אחסון של מידע בכמות גדולה. - כל יישומי המחשב צריכים לאחסן מידע כלשהו. כאשר תהליך רץ הוא יכול לאחסן כמות מוגבלת של מידע, במרחב הכתובת שלו. למספר יישומים זה מספיק, אולם לאחרים, כמו חברות תעופה, בנקים, המקום קטן מידי.
ב. המידע חייב להישאר גם לאחר שהתהליך שהשתמש בו הסתיים. -
הבעיה השניה של שמירת אינפורמציה בכתובת התהליך היא שכאשר התהליך מסתיים, המידע אובד.
ג. יש לאפשר לכמה תהליכים במקביל להשתמש במידע.- אם המידע מאוחסן אצל התהליך, רק הוא יכול לפנות אליו, ובחברת טלפונים המשמעות היא שניתן להתבונן במספר אחד בכל פעם.
הפתרון לדרישות אלה היא אחסון מידע בדיסקים ושאר אמצעי אחסון חיצוניים ביחידות הקרויות קבצים – FILES.

עמוד 146
הקובץ הוא מבנה וירטואלי (מדומה). המימוש הפיסי שלו שונה לחלוטין מזה שהמשתמש רואה לנגד עיניו.
המבנה הלוגי המאגד את הכל הקבצים במערכת נקרא FILE SYSTEM, דם מערכת הקבצים כולה היא מבנה מדומה, והמשתמש אינו צריך לדעת דבר על אופן מימושה הפיזי. בדיון שלנו נתייחס למערכת הקבצים כאל חלק ממערכת ההפעלה במערכת מתקדמות הגישה היא לבצע הפרדה כאשר כל הניהול והיישום של מערכת הקבצים מתבצע באמצעות תהליך שרץ מחוץ לגרעין של מערכת ההפעלה. תהליך זה פותח, סוגר, קורא או כותב קבצים לפי בקשות המופנות אליו מתוך גרעין מערכת ההפעלה. עם מילוי הבקשה, התהליך מעביר את התשובות לבקשות אל תיבת דואר שנקבעה מראש.

4.1 קבצים
הסעיפים הבאים עוסקים במערכת הקבצים כפי שהיא נראית מנקודת מבטו של המשתמש. קבצים מספקים דרך לאחסן מידע על הדיסק ולקרוא אותו מאוחר יותר. איך ואיפה המדע מאוחסן – מוסתר מהמשתמש.

4.1.1 שמות לקבצים
אחד המאפיינים החשובים ביותר של מכונה מופשטת הוא הדרך בה האובייקטים המנוהלים נקראים. כאשר תהליך יוצר קובץ הוא נותן לו שם, וכאשר התהליך מסתיים הקובץ עדיין נשאר ויכול לשרת תהליכים אחרים.
החוקים המדויקים של מתן שמות לקבצים משתנים ממערכת למערכת, אך קיים מבנה משותף ביניהם : שם הקובץ צריך להכיל לא יותר מששמונה אותיות גדולות. זוהי התת קבוצה של שמות הקבצים המוכרת על ידי כל המחשבים הקיימים היום.
MS-DOS אינה מבחינה בין אותיות קטנות לגדולות.
UNIX מבחינה בין אותיות קטנות לגדולות.

עמוד 147
מערכות הפעלה רבות תומכות בשני חלקים לשם הקובץ המופרדים בנקודה – שם פרטי ושם משפחה (החלק שאחרי נקרא FILE EXTENSION. שמות המשפחה משמשים זיהוי לסוג הקובץ כאשר ישנן סיומות שנאכפות כגון BAT. וישנן סיומות שמקלות על המשתמש כגון TEXT.

עמוד 148
לעיתים ההפרדה חיונית, לדוגמא ניתן לתת למהדר C, להדר קבצים בשפת C וקבצים באסמלי, ולכן הסיומת (C.) חיונית למהדר על מנת לדעת איזה סוג קובץ.

4.1.2 המבנה הפנימי של הקובץ

מערכות הפעלה כלליות כגון UNIX ו- MS-DOS אינן מניחות דבר לגבי המבנה הפנימי של קובץ. מבחינתן קובץ הוא רצף של ביטים בלבד. האחריות למבנה הקובץ מוטלת על התהליך או על המשתמש עצמו. עובדה מגבירה את הגמישות ומאפשרת למשתמש לאחסן כל דבר בקובץ ולתת לו שם לפי נוחותו.
הצעד הראשון מוסבר באיור 4.2. במודל זה קובץ הוא רצף באורך קבוע , כל אחד בעל מבנה פנימי אחר. אם כי דרישת גודל קבוע משתנה בין המערכות (יש כאלו הרואות קובץ כגודל משתנה).

עמוד 149

מבנה נוסף הוא קובץ המורכב מעץ של רשימות, לא כולן באותו אורך, כל אחת מכילה שדה מפתח במקום קבוע ברשימה. העץ ממוין על ידי שדה המפתח, ומאפשר חיפוש של מפתח מסוים.
הפעולה הבסיסית כאן היא לא לקבל את הרשימה "הבאה", וזה אפשרי, אלא לקבל רשימה עם מפתח מסוים, כך שאם נבקש קובץ עם סיומת מסוימת נקבל אותו מבלי קשר למיקומו. יותר מכך מערכת ההפעלה יכולה להוסיף רשימות לקובץ כאשר מערכת ההפעלה, ולא המשתמש מחליטה היכל למקם אותה.
סוג זה של קובץ קצת שונה מרצף לא מובנה של ביטים הנמצא בשימוש UNIX ו--MS DOS.
מערכת ההפעלה מספקת מנגנון אך המדיניות (תוכן, סידור בספריות) נותרת בידי התהליכים.
ישנם קבצים שהמבנה שלהם נבחן ישירות על ידי מערכת ההפעלה:

א. קבצים המכילים מידע פנימי של מערכת ההפעלה.
ב. מדריכים (DIRECTORIES).
ג. קבצים המיועדים להרצה.

4.1.3 טיפוסי קבצים

UNIX ן- MS-DOS תומכות בטיפוסים אלה של קבצים:

א. קבצים רגילים – REGULAR FILE: מכילים אינפורמציה של המשתמש.
ב. ספריות – DIRETORIES: קבצי מערכת המשמרים את מבנה מערכת הקבצים.
ג. קבצי תווים מיוחדים CHARACTER SPECIAL FILE: מתייחסים לקלט\פלט ומשמשים לעיצוב התקני קלט\פלט כמו מסופים, מדפסות ורשתות.
ד. BLOCK SPECIAL FILES: משמשים לעיצוב דיסקים.

אנו נעסוק בקבצים רגילים בעיקר.
רוב הקבצים הרגילים הם קבצי אסקי (ASCII) או קבצים בינריים. קבצי אסקי מכילים שורות של טקסט + מאפיינים למעבר שורה (CARRIAGE RETURN, LINE FEED). השורות לא צריכות להיות באותו אורך.
יתרון קבצי האסקי הוא שניתן להציגם ולהדפיסם כפי שהם וכן ניתן לערוך אותם על ידי תוכניות עריכה סטנדרטיות. תוכניות רבות משתמשות בקובצי אסקי עבור קלט\פלט. וקל לקשר את הפלט של תוכנית אחת לקלט של תוכנית אחרת, כמו צינורות המעטפת. כלומר יתרון נוסף הוא שהם ניתנים להעברה בין תוכניות שונות ומחשבים שונים.
הקבצים האחרים הם קבצים בינארים, שמשמעותם שהם אינם קובצי אסקי. הדפסתם תיתן משהו משונה מאוד. תוכנית הפותחת קובץ מסוג זה אמורה להכיל בתוכה את כל המידע הדרוש כדי להשתמש בו. מערכת UNIX לדוגמה בודקת רק את המבנה הפנימי של הקבצים. אומנם באופן טכני הקבצים הם רק רצף של ביטים, אולם מערכת ההפעלה תבצע קבצים בעלי מבנה נכון.

עמוד 150

יש בו חמישה חלקים: כותרת, טקסט, נתונים, ציון מיקום, וטבלת סמלים.
הכותרת מתחילה עם MAGIC NUMBER, המזהה את הקובץ כקובץ לביצוע. אחר כך ישנם מספרים בני 16 ביטים, המלמדים על גודלם של חלקים שונים בקובץ, על הכתובת לתחילת הביצוע, וכמה דגלי ביטים. אחרי הכותרת ישנם הטקסט והנתונים של התוכנית

עצמה. הם נטענים לזיכרון, וממוקמים תוך שימוש בביטים למיקום (RELOCATION BITS). טבלת הסמלים משמשת לאיתור שגיאות (DEBUGGING).
דוגמה נוספת לקובץ בינארי היא ארכיון. הוא מורכב מאוסף של ספריות של פרוצדורות (מודולים), מהודרים אולם לא מקושרים. כל אחד פותח כותרת המכילה את שמו, תאריך יצירתו, בעלים, קוד הגנה וגודל. כמו בקובץ לביצוע, כותרת המודול מלאה במספרים בינארים. העתקתם למדפסת תפיק זבל.
כל מערכות ההפעלה צריכות לזהות את סוג הקובץ, את קובצי הביצוע שלהם, ויש כאלו שמזהים יותר. יש מערכות הבודקות את תאריך יצירת הקובץ בכל קובץ העומד לביצוע. אחר כך הן מאתרות את הקובץ המקורי לבדוק האם עבר שינוי. אם כן, מייד מהדרים את המקור.

עמוד 151

הבעיה נוצרת כאשר משתמש עושה משהו שלא נצפה על ידי מתכנני המערכת.
העתקת file.dat ל- file.pas תדחה על ידי מהדר פסקל (על מנת להגן על המשתמש משגיאות).

4.1.4 גישה לקבצים

מערכות ההפעלה הישנות סיפקו רק גישה אחת לקבצים : גישה סדרתית. לפי גישה זו תהליך יכל לקרוא בתים או רשומות מקובץ לפי סדר, החל מתחילת הקובץ ולא יכל לדלג אל בתים. גישה סדרתית מתאימה להתקני חומרה כמו טייפ מגנטי, אולם לא לדיסק.
כאשר משתמשים בדיסק לאחסון קבצים, יש אפשרות לקרוא קבצים שלא לפי הסדר, או להיכנס לרשומה לפי מפתח, במקום לפי המיקום. קבצים שניתן לקראם שלא לפי הסדר נקראים RANDOM ACCESS FILES. גישה אקראית.
הגישה האקראית חיונית במיוחד לבסיסי נתונים בהם יש לשלוף רשומות מתוך כלל הנתונים. אם לקוח חברת תעופה מחפש נתון על טיסה כלשהי, צריכים את האפשרות לשלוף אותו מייד, ולא להתחיל לעבור על כל רשומות הטיסות כדי למצוא אותו.
ישנן שתי שיטות לספק את המקום ממנו יש להתחיל לקרוא :

א. פעולת READ – בכל פעם שמשתמשים בה היא מספקת את המיקום בקובץ ממנו יש להתחיל לקרוא.

ב. פעולת SEEK – מאתחלת את המיקום הנוכחי שאחריו ניתן לקרוא באופן סדרתי.

במערכות ישנות יותר מסווגים קבצים בעת יצירתם כסדרתי או אקראי. הדבר מאפשר למערכת ההפעלה לספק אמצעי אחסון שונים עבור שתי המחלקות.
במערכות המודרניות לא משתמשים בשתי מחלקות. כל הקבצים באופן אוטומטי, הם בעלי גישה אקראית.

4.1.5 הצמדת תכונות לקבצים

לכל קובץ יש שם ונתונים. בנוסף כל מערכת הפעלה מצמידה מידע נוסף לכל קובץ, כגון התאריך והשעה שבו הוא נוצר, וגודל הקובץ. מידע נוסף זה קרוי – FILE'S ATTRIBUTES. רשימת התכונות הנוספות משתנה ממערכת הפעלה אחת לאחרת.
ארבעת התכונות הראשונות קשורות להגנת הקבצים, ומלמדות מי יכול להיכנס לקובץ ומי לא.

עמוד 152

להלן רשימת התכונות :

1. הגנה – מי יכול להיכנס ומי לא.
2. סיסמא – הסיסמא דרושה כדי להיכנס לקובץ.
3. CREATOR – מספר זיהוי של האדם שיצר את הקובץ.
4. בעלים – הבעלים הנוכחי.
5. דגל READ ONLT – 1 קריאה בלבד, 0 לקריאה ולכתיבה.
6. דגל מוסתר – 0 רגיל, 1 כדי לא להציג ברשימת הקבצים.
7. דגל המערכת – 0 לקובץ רגיל, 1 לקובץ מערכת.

8. דגל ארכיון – 0 עבר גיבוי, 1 יש לגבות.
9. דגל בינארי\אסקיי - 0 לקובץ אסקיי, 1 לקובץ בינארי.
10. דגל לגישה אקראית – 0 לגישה סדרתית בלבד, 1 לגישה אקראית.
11. דגל זמני – 0 רגיל, 1 למחיקה עם סיום התהליך.
12. דגלי נעילה – 0 לא לנעילה, מספר שונה מ- 0 לנעילה.
13. אורך רשומה – מספר הביטים ברשומה.
14. מיקום המפתח – ההיסט של המפתח בכל רשומה.
15. אורך המפתח – מספר הבתים בשדה המפתח.
16. זמן הווצרות – תאריך וזמן בו נוצר הקובץ.
17. תאריך הכניסה האחרונה – תאריך וזמן הכניסה האחרונה לקובץ.
18. זמן שינוי – תאריך וזמן בו קובץ עבר שינוי בפעם האחרונה.
19. גודל נוכחי – מספר הבתים שבקובץ.
20. גודל מקסימלי – הגודל המקסימלי אליו יכול הקובץ לגדול.

הדגלים הם ביטים או שדות קצרים השולטים או מאפשרים תכונה מסוימת. לדוגמא, לא להופיע ברשימת הקבצים. דגל הארכיון הוא ביט שמסמן האם קובץ עבר גיבוי או לא. התוכנית המגבה מאפסת אותו, ומערכת ההפעלה מדליקה אותו בכל פעם שקובץ עובר שינוי. בדרך זו תוכנית גיבוי יכולה לאתר קבצים הזקוקים לגיבוי. דגל זמני מסמן קובץ למחיקה אוטומטית, כאשר התהליך שיצר אותו מסתיים.
אורך רשומה, מיקום מפתח, ושדה אורך המפתח נמצאים רק בקבצים אשר ניתן לסרוק את רשומותיהם תוך שימוש במפתח. הם מספקים את המידע הדרוש למציאת המפתחות.
משתני הזמן מסמנים מתי קובץ נוצר, תאריך הכניסה האחרונה לקובץ, ותאריך השינוי האחרון בו. הם שימושיים למגוון יישומים.
הגודל הנוכחי מציג את גודל הקובץ כרגע. ישנם מערכות הפעלה מסוימות הדורשות לציין את הגודל המקסימלי שאליו יכול להגיע הקובץ, זאת על מנת לשמור לו מקום אחסון, מראש.

עמוד 153

4.1.6 פעולות על קבצים

קבצים קיימים על מנת לאחסן מידע, ולאפשר לשלפו מאוחר יותר. מערכות שונות מספקות פעולות שונות, כדי לאפשר אחסון ושליפה. להלן קריאות המערכת הנפוצות ביותר הקשורות בקבצים:

1. CREATE – הקובץ נוצר ללא נתונים. הסיבה לקריאה היא להודיע על הקובץ, ולעדכן כמה מתכונותיו.
2. DELETE – כאשר אין צורך יותר בקובץ, יש למחקו כדי לפנות מקום על הדיסק. תמיד קיימת קריאת מערכת למטרה זו. בתוספת, ישנן מערכות הפעלה המוחקות באופן אוטומטי קבצים שלא היו בשימוש במשך N ימים.
3. OPEN – לפני שימוש בקובץ, על תהליך לפתוח אותו. המטרה לקריאת OPEN היא לאפשר למערכת להביא את התכונות, ורשימה של כתובות מהדיסק אל הזיכרון הראשי עבור כניסה מהירה בקריאות עוקבות.
4. CLOSE – כאשר כל הכניסות הסתיימו, אין צורך בתכונות, ובכתובות בדיסק, כך שיש לסגור את הקובץ, כדי לפנות מקום בתוך הטבלה הפנימית. מערכות רבות מעודדות זאת על ידי הגבלת המספר המקסימלי של קבצים פתוחים, בתהליכים.
5. READ- קריאת נתונים מהקובץ. בדרך כלל הקריאה נעשית מהמיקום הנוכחי. על התהליך הקורא לציין לכמה נתונים הוא זקוק, ולספק חוצץ לקליטתם.
6. WRITE – כתיבת נתונים אל הקובץ, שוב, בדרך כלל במיקום הנוכחי. עם המיקום הנוכחי הוא סוף הקובץ, אזי הקובץ גדל. אם המיקום הנוכחי הוא באמצע הקובץ, נתונים קיימים נמחקים, ואובדים.
7. APPEND – קריאה זו היא כתיבה מצומצמת. היא מאפשרת רק להוסיף נתונים לסוף הקובץ. מערכות המספקות מספר מינימלי של קריאות מערכת, לא

מכילות בדרך כלל את APPEND, אולם מערכות רבות מספקות אפשריות מקבילות לביצוע אותו דבר.

8. SEEK – עבור כניסה אקראית לקבצים, יש צורך בשיטה כדי לציין מהיכן להוציא את הנתונים. קריאת מערכת SEEK, ממקמת את המצביע מהמיקום הנוכחי, אל מקום ספציפי בקובץ. לאחר השלמת הקריאה, ניתן לקרוא נתונים, או לכתוב נתונים למקום זה.
9. GET ATTRIBUTES – לעיתים קרובות תהליכים נזקקים לתכונות הקובץ תוך כדי ביצוע. (לדוגמא, האם הקובץ עבר שינוי ועוד).
10. SET ATTRIBUTES - כמה מתכונות הקובץ ניתנות לסימון על ידי המשתמש, ויכולים להשתנות לאחר יצירת הקובץ. קריאת מערכת זו מאפשרת זאת. למשל תכונות הגנה על הקובץ, וגם כמה מהדגלים.
11. RENAME – מאפשר לשנות שם קובץ קיים. לא תמיד יש צורך בכך היות ותמיד ניתן להעתיק קובץ, אל קובץ חדש בשם הדרוש, ואז למחוק את הקובץ הישן.

עמודים 154 155

תוכנית לדוגמא המשתמשת בקריאות מערכת לקבצים.

עמוד 156

4.1.7 מיפוי קבצים לזיכרון

אצל אנשים רבים קיימת התחושה שכניסה לקבצים, כפי שהוסבר היא מסורבלת קמעה, ולא נוחה, במיוחד בהשוואה לכניסה לזיכרון הרגיל. מסיבה זו כמה מערכות הפעלה מספקות דרך למפות קבצים לתוך מרחב הכתובות של תהליך בהרצה. לשם כך ישנן שתי קריאות מערכת MAP ו- UNMAP.

MAP – נותנת לקובץ שם וכתובת וירטואלית, ומערכת ההפעלה שמה ישירות את הקובץ בכתובת זו (החל מכתובת זו והלאה). לדוגמא, נניח שקובץ F, שאורכו 64K ממופה אל כתובת וירטואלית 512K. אזי כל הוראת מחשב הקוראת את תוכן הבית בכתובת 512K, מקבלת את בית 0 של הקובץ. אם נכתוב לכתובת ' 512k+1100 אזי בית 1100 ישתנה בקובץ. כאשר התהליך מסתיים, הקובץ שעבר שינוי מושאר בדיסק, כאילו עבר שינוי על ידי שילוב של קריאות מערכת SEEK ו- WRITE.

מה שבאמת קורא הוא שהטבלאות הפנימיות של המערכת משתנות, כדי להציג את הקובץ כאחסון תומך לקובץ. לפיכך, קריאה מכתובת 512K הגורמת לפסיקת דף, מכניסה את דף 0 של הקובץ. בדומה, כתיבה ל- 512K+1100 הגורמת לפסיקת דף, מכניסה את הדף המכיל את הכתובת, לאחר מכן ניתן לכתוב לזיכרון. אם הדף מפונה על ידי אלגוריתם להחלפת דפים, הוא נכתב שוב במקום המתאים בקובץ. כאשר התהליך מסתיים, כל הדפים ששונו נכתבים חזרה לקבצים שלהם.

עמוד 157

מיפוי קבצים פועל היטב במערכות התומכות בסגמנטציה, שם כל קובץ יכול להיות ממופה לסגמנט משלו כך שהבית ה-K בקובץ הוא הבית ה-K בסגמנט. באיור 4-6 אנו רואים תהליך עם שני סגמנטים, טקסט ונתונים. נניח שהתהליך מעתיק קובץ אחד לאחר, כמו התוכנית באיור 5-4. תחילה הוא ממפה את קובץ המקור, נאמר abc אל סגמנט. אחר כך הוא יוצר סגמנט ריק וממפה אותו אל קובץ היעד, xyz, בדוגמא שלנו. בשלב זה התהליך יכול להעתיק את סגמנט המקור אל סגמנט היעד, תוך שימוש בלולאת העתקה רגילה. אין צורך בקריאות מערכת READ AND WRITE. בסיום, ניתן לבצע את קריאת מערכת UNMAP כדי להעביר את הקבצים ממרחב הכתובות, ואז לצאת. קובץ הפלט xyz, קיים עתה, כאילו נוצר בדרך הקונבנציונלית.

כלומר היתרון המשמעותי של המיפוי הוא – מהירות, היות ומיפוי חוסך את הצורך בקלט\פלט.

חסרונות מיפוי קבצים בזיכרון:

1. בעיית הגודל הצפוי של הקובץ – קשה לדעת בדיוק מה מספר הבתים שנכתבו בכל דף, לכן המערכת יכולה לדעת בביטחון רק את גודל הדף.

2. הקושי בשיתוף קבצים בין מספר תהליכים בו-זמנית – כאשר קובץ ממופה על ידי תהליך אחד אך נפתח לקריאה על ידי תהליך שני. אם התהליך הראשון משנה דף מסוים, השינוי לא יהיה בקובץ שעל הדיסק (זה שהתהליך השני קורא ממנו) עד שהדף לא יתפנה. המערכת צריכה לדאוג ביתר זהירות ששני תהליכים לא יראו שתי גרסאות שונות של אותו קובץ.
3. בזבוז שטחי זיכרון – שיכולים לשמש להרצת תהליכים (הורדת המקביליות).
4. הקובץ יכול להיות גדול יותר מגודל הסגמנט – או אפילו ממרחב הכתובות הוירטואליות. מיפוי חלקי של הקובץ אינו יעיל כמו מיפוי הקובץ כולו.

עמוד 158

4.2 מדריכים – DIRECTORIES

בכדי שאפשר יהיה לעקוב אחרי הקבצים – מערכת הקבצים מספקת מדריכים. ברוב מערכות הקבצים, המדריכים גם הם עצמם קבצים. זאת כדי לאפשר את המבנה האופייני של עץ. בסעיפים הבאים יסקרו פעולות הקשורות למדריכים.

4.2.1 עצי קבצים ומסלולים לקבצים

מדריך מכיל מספר כניסות, אחת לכל קובץ. ישנם שני מבנים נפוצים עיקריים:

1. כל כניסה מכילה את שם הקובץ, תכונות הקובץ והכתובת על הדיסק, במקום שהנתונים מאוחסנים.
2. הכניסה מכילה את שם הקובץ ומצביע למבנה נתונים אחר שבו נמצאים אפיוני (תכונות) הקובץ, והכתובת על הדיסק.

כאשר קובץ נפתח, מערכת ההפעלה סורקת את המדריך שלו עד למציאת שם הקובץ שיש לפותחו. אחר כך היא מעתיקה את אפיוני הקובץ, והכתובות בדיסק, (או ישירות מהכניסה במדריך, או ממבנה הנתונים המוצבע מהכניסה), ומכניס אותם לטבלה בזיכרון הראשי. מספר המדריכים משתנה ממערכת למערכת.

שלושה מודלים למערכת הקבצים:

1. התכנון הפשוט ביותר הוא עבור המערכת לשמירת מדריך אחד הכולל את כל קובצי המשתמש, כפי שמוסבר באיור 4-8. כאשר ישנם הרבה משתמשים, הנותנים את אותם שמות לקבצים, המערכת תגיע למצב של קונפליקט, ולא יהיה ניתן לעבוד בה. מודל זה בשימוש רק במערכת הפעלה פרימיטיביות.
2. ספריה לכל משתמש – תכנון זה מונע קונפליקט שמות, בין משתמשים, אולם לא משביע רצון עבור משתמש בעל קבצים רבים. די נפוץ בין משתמשים לקבץ את הקבצים שלהם בדרך לוגית כלשהי.
3. עץ ספריות - כך שלכל משתמש יש ספריה משלו עם תתי ספריות, ולמעשה עץ ספריות משלו. היתרון הוא שמבנה העץ הוא רקורסיבי. קל מאוד למזג עצים שונים וכך להסתיר את העובדה כי חלקים של העץ עשויים להיות על התקן פיסי שונה (שקיפות זו לא קיימת ב-MS-DOS היות והמסלול לקובץ כולל את שמו הלוגי של ההתקן, למשל :A).

עמוד 159

4.2.2 שמות מסלולים

כאשר מערכת הקבצים מאורגנת כעץ מדריכים, יש צורך בדרך לציין את שם הקובץ. ישנן שתי שיטות עיקריות נפוצות:

1. ABSOLUTE PATH NAME - המורכב מהמסלול משורש העץ אל הקובץ המבוקש. בדרך זו מתואר המסלול המלא מהשורש כולל כל תתי הספריות בדרך. ב-UNIX מרכיבי המסלול מופרדים ב- /, ב-MS-DOS הם מופרדים ב-\

   וב- MULTICS ב- >. לא משנה איזה תו נמצא בשימוש, אם הוא נמצא בתחילת שם המסלול, אזי המסלול הוא אבסולוטי.
2. RELATIVE PATH NAME – המסלול נרשם ביחס לספריית העבודה, כלומר הספרייה שכרגע אנו נמצאים בה. המשתמש יכול לציין ספריה אחת, בספריית

העבודה הנוכחית, בכל מקרה כל מסלול שלא מתחיל בשורש העץ מחושב באופן יחסי לספריית העבודה.

עמוד 160

לדוגמא אם ספריית העבודה הנוכחית הוא user/ast/ אזי ניתן לפנות לקובץ שמסלולו האבסולוטי הוא user/ast/mailbox/ פשוט כ-mailbox, במילים אחרות, פקודת UNIX הבאה:

CP /user/ast/mailbox/usr/ast/mailbox.bak והפקודה CP mailbox mailbox.bak הן זהות, אם

ספריית העבודה היא user/ast/. כאשר המסלול היחסי נוח יותר.

תוכניות מסוימות שצריכות להיכנס לקובץ מסוים מבלי קשר לספריית העבודה. במקרה זו עליהן תמיד לציין את שם המסלול האבסולוטי. המסלול האבסולוטי תמיד יעבוד, לא משנה מהי ספריית העבודה.

אם תוכנית מסוימת זקוקה נניח לקבצים רבים מספרית user/lib/ גישה אלטרנטיבית היא לאשר קריאת מערכת המשנה את ספרית העבודה ל- user/lib/, ואז לציין את שם הקובץ כפרמטר הראשון עבור open. שינוי ספרית העבודה מאפשר לו לדעת היכן הוא נמצא בעץ המדריכים והוא יכול להשתמש במסלול היחסי.

ברוב המערכות לכל תהליך יש את ספריית העבודה שלו, כך שכאשר תהליך שינה את ספריית העבודה שלו ואחר כך יצא, אף תהליך אחר אינו מושפע מכך ולא נשארות עקבות במערכת הקבצים. בדרך זו זה בטוח עבור תהליך לשנות ספריית עבודה. (כלומר ספרית העבודה ב-SHELL לא משתנה, אלא באופן זמני בתהליך עצמו). מצד שני אם פרוצדורת ספריה משנה את ספרית העבודה שלה, ולא משנה אותה בחזרה, יתכן ששאר התוכנית לא תעבוד, היות וההנחה לגבי מיקום הפרוצדורה עלולה להיות שגויה כעת. מסיבה זו פרוצדורת ספריה משנות רק לעיתים רחוקות את ספרית העבודה, וכאשר הן מוכרחות, הן תמיד חוזרות לספריה המקורית, בסיום.

לרוב מערכות ההפעלה תומכות מערכת היררכית של ספריות יש שתי כניסות מיוחדות בכל ספריה, "." מכונה DOT ומתייחס לספריה הנוכחית. ".." מכונה DOTDOT ומתייחס לספרית ההורה של הספרייה הנוכחית.

לדוגמא עץ קבצים ב-UNIX (ראה איור 4-9). לתהליך מסוים יש כספרית עבודה את user/ast/ . הוא יכול להשתמש ב- .. כדי לעלות למעלה בעץ. לדוגמא, הוא יכול להעתיק קובץ user/lib/dictionary/ אל הספריה שלו תוך שימוש בפקודת מעטפת CP .. /lib/dictionary . המסלול הראשון מורה למערכת להתייחס אל ספרית user, ואז לרדת לספרית lib ולמצוא את הקובץ dictionary.

הארגומנט השני הוא שם הספרייה הנוכחית.. כאשר פקודת CP מקבלת שם ספריה (כולל DOT) כארגומנט האחרון, היא מעתיקה את כל הקבצים שם. כמובן דרך רגילה יותר לבצע העתקה היא: CP /user/lib/dictionary/ . כאן השימוש ב- DOT חוסך למשתמש את הקלדת dictionary בפעם השניה.

עמוד 161

4.2.3 פעולות על מדריכים

1. CREATE – יוצר ספריה. הספרייה ריקה למעט DOT ו- DOTDOT, הנמצאות באופן אוטומטי.
2. DELETE – מחיקת ספריה. ניתן למחוק ספריה ריקה בלבד, כאשר ספריה המכילה רק DOT ו- DOTDOT נחשבת ריקה.
3. OPENDIR – קריאת ספריה. ניתן לקרוא את רשימת כל הקבצים המוכלים בספריה. לפני שניתן לקרוא בספריה, יש לפתוח אותה.

עמוד 162

4. CLOSEDIR – עם סיום קריאה מספריה, יש לסגור אותה על מנת לפנות מקום בטבלה הפנימית.

5. READDIR – הקריאה מחזירה את הכניסה הבאה בספריה הפתוחה. באופן פורמלי ניתן היה להשתמש בקריאת מערכת READ, אולם בגישה זו מכריחים את המתכנת להיות מודע למבנה הפנימי של הספריות ולעבוד בהתאם.
6. RENAME – שינוי שם הספרייה (כמו שינוי שם קובץ).
7. LINK – קישור זוהי טכניקה המאפשרת לקובץ להופיע ביותר מספריה אחת. קריאת מערכת זו מציינת שם קובץ, ושם מסלול, ויוצרת LINK לקובץ הנ"ל. בדרך זו הקובץ מופיע בכמה ספריות במקביל.
8. UNLINK – הסרת כניסה לספריה. לאחר הביצוע הקובץ נמצא בספריה אחת בלבד. אם לפני ביצוע הפקודה הוא נמצא בכמה ספריות, רק המסלול שמצוין מוסר. השאר נשארים. ב-UNIX קריאת מערכת למחיקת קבצים, היא למעשה UNLINK.

4.3 יישום מערכת הקבצים

בסעיפים הבאים נעסוק במבנה מערכת הקבצים מנקודת מבטו של המתכנן. את הממשק של המערכת כבר הגדרנו. מה שנותר הוא לממשו ביעילות ובבטיחות.

4.3.1 יישום קבצים

נושא המפתח ביישום אחסון קבצים הוא לבצע מעקב אחר איזה בלוק בדיסק מחזיק איזה קובץ. נפרט שיטות אחדות:

עמוד 163

הקצאה רצופה

אחסון כל קובץ כרצף בלוקים בדיסק. אם כל בלוק הוא 1K והקובץ 50K – יהיו 50 בלוקים רצופים בדיסק עבור אחסון קובץ זה. יתרונות:

א. קל ליישום – פשוט לקיים מעקב מכוון שצריך לזכור רק את הכתובת על הדיסק של הבלוק הראשון.

ב. הביצועים מרשימים – כל הקובץ יכול להיקרא בפעולה אחת בשל הרצף שבו הוא מאוחסן. (שום אלגוריתם אחר לא מתקרב לביצועים אלה).

חסרונות:

א. אינו בר-ביצוע – אלא אם כן גודלו המקסימלי של הקובץ ידוע בעת יצירתו. בלי מידע זה מערכת ההפעלה לא תדע כמה מקום להקצות על הדיסק.

ב. נוצרים מצבי FRAGMENTATION רבים – כאשר נשארים בדיסק קטעים קטנים רבים, מבוזבזים. כלומר הוא מבזבז מקום שיכל להיות בשימוש. מיזוג שטחים בדיסק לוקח זמן רב ולכן יקר. אם כי ניתן לעשות זאת בלילות כאשר המערכת לא עובדת.

השיטה מתאימה למערכות הדורשות פעולות קלט\פלט מהירות ביותר, אשר גודל הקבצים בהן ידוע מראש.

רשימה משורשרת

אחסון כל קובץ ברשימה מקושרת של בלוקים (שרטוט 10-4). השדה הראשון בכל בלוק הוא מצביע לבלוק הבא ברשימה, והשאר לנתונים. יתרונות:

א. ניתן להשתמש בכל בלוק בדיסק בכל גודל.

ב. אין מקום מבוזבז (אין חורים).

ג. יעיל יותר כאשר הספרייה מאחסנת רק את הכתובת של הבלוק הראשון ברשימה.

חסרונות:

א. הגישה לקבצים איטית – בגישה אקראית. בגישה סדרתית היא מהירה.

ג. לא ניתן לאחסן נתונים בחזקה של 2 מכיוון שהמצביע תופס מספר בתים. זה פחות יעיל מכיוון שתוכניות רבות קוראות וכותבות נתונים בגדלים של חזקה של 2.

עמוד 164

רשימה משורשרת הממומשת בעזרת אינדקס

רשימה הממומשת במערך. ניתן להתגבר על שני החסרונות שלעיל על ידי שמירת ההצבעות בטבלה מרוכזת בעלת אינדקסים (מספרים פיזיים של בלוקים), כך שהטבלה מכילה את האינדקס הפיזי של הבלוק הבא. (שרטוט 11-4). יתרונות:

א. כל הבלוק פנוי לאחסון נתונים (פותר חיסרון ב׳ של רשימה משורשרת).
ב. הגישה האקראית ניתנת ליישום באופן מהיר וקל יותר.
ג. למרות שעדיין יש למצוא את ההיסט בתוך הקובץ, הטבלה נמצאת בזיכרון כך שאין צורך בגישות רבות לדיסק.
ד. כמו הגישה הקודמת הספרייה מחזיקה רק את האינדקס הראשון של כל קובץ (MS-DOS משתמשת בשיטה זו).

החסרון העיקרי הוא שיש לשמור את הטבלה בזיכרון כך שהיא תופסת מקום רב בזיכרון.

עמוד 165

שיטת ה- I-NODES

זוהי השיטה המשמשת ב-UNIX, והיא באה להקטין את הטבלה המרכזית שחייבים לשמור בזיכרון, תוך שמירה על היתרונות של שיטת האינדקס. שים לב כי באופן עקרוני הגודל המקסימלי של קובץ חסום, אך למעשה גודל זה מספיק לכל צורך מעשי. להלן פרוט השיטה:

לכל קובץ יש טבלה קטנה משלו הנקראת I-NODE (INDEX NODE) אשר מאחסנת את התכונות של הקובץ ואת הכתובות של הבלוקים שלו שעל גבי הדיסק (שרטוט 12-4). אם הקובץ קטן אז המספר המצומצם של הכתובות מאוחסן בטבלת ה- I-NODE אשר נטענת לזיכרון כאשר הקובץ נפתח.

עבור קבצים גדולים יותר, טבלת ה-I-NODE מכילה כתובת אחת של בלוק בדיסק הנקרא SINGLE INDIRECT BLOCK. שהוא זה שמכיל את כתובות הקובץ בדיסק. אם זה עדיין לא מספיק אזי מחזיקים טבלאות נוספות.

UNIX משתמשת במבנה זה.

4.3.2 יישום מדריכים

לפני שניתן לקרוא קובץ יש לפתוח אותו. כאשר קובץ נפתח, מערכת ההפעלה משתמשת במסלול שהמשתמש נתן בכדי להגיע לכניסה הנכונה בספריה. הכניסה בספריה מספקת את המידע הדרוש למציאת הבלוקים בדיסק. המידע יכול להיות כתובת בדיסק של כל הקובץ (בהקצאה רצופה), או מספרו של הבלוק הראשון (רשימה מקושרת), או מספר ה- I-NODE. בכל המקרים הפעולה העיקרית של מערכת הספריות היא למפות את שם הקובץ (באסקי) לאינפורמציה הדרושה על מנת לאתר את הנתונים.

נושא הקשור לכך היא היכן יש לאחסן את אפיוני הקובץ.

עמוד 166

רוב המערכות מאחסנות אותם ישירות בכניסה של הקובץ בספרייה. מערכות המשתמשות ב- I-NODE יכולות לאחסן את התכונות ב-I-NODE , במקום בכניסה של הקובץ בספריה. כפי שנראה מאוחר יותר, לשיטה זו יש יתרונות מסוימים על האחסון בכניסה בספריה.

מדריכים במערכת ההפעלה CP/M

לדיון זה חשיבות היסטורית בלבד שכן מערכת זו, שהפעילה מחשבים אישיים רבים, איננה בשימוש עוד. המדריך כאן היה פשטו בצורה קיצונית, והיה רק אחד כזה.

במערכת זו הייתה רק ספריה אחת, כך שחיפוש אחר קובץ היה פשוט חיפוש תחת ספריה זו. כאשר מוצאים את הכניסה בה יש את כל הכתובות של הבלוקים בדיסק, וכל תכונות הקובץ. אם לקובץ יש יותר בלוקים בדיסק מאשר לכניסה אחת, הקובץ מקבל כניסות נוספות בספריה. (שרטוט 13-4). לשדות באיור יש את המשמעות הבאה: שדה ה- USER CODE עוקב האם המשתמש הוא בעל הקובץ. במשך החיפוש רק הכניסות השייכות למשתמש זה נסרקות. שני השדות הבאים מספקים את שם והרחבות (הרשאות) הקובץ.

שדה EXTEN דרוש היות ואם קובץ גדול מ- 16 בלוקים, הוא תופס כמה כניסות בספריה. שדה זה אומר לנו מהו סדר הכניסות. 16 השדות הבאים אומרים לנו איזה מ16 הבלוקים בדיסק נמצאים בשימוש, ומכילים את הכתובות של הבלוקים בדיסק. המערכת לא יודעת את גודל הקובץ בבתים, אלא בבלוקים.

מדריכים במערכת ההפעלה MS-DOS

הכניסה בספריה מורכבת משם הקובץ, הרחבות (סוג הקובץ), תכונותיו, מספר האינדקס בטבלה של הבלוק הראשון בדיסק, ועוד. שרטוט 4-14.
כל ספריה יכולה להכיל ספריות נוספות כך שלמעשה כל מדריך הוא צומת בעץ הקבצים.
מה שמאפשר את קיומו של עץ קבצים ב-MS-DOS היא העובדה כי מדריך יכול להכיל לא רק פרטים על קבצים, אלא גם פרטים על מדריכים נוספים.

עמוד 167

מדריכים במערכת ההפעלה UNIX
מבנה הספריות הוא פשוט מאוד, ראה שרטוט 4-15. כל כניסה של קובץ בספריה מכילה רק את שם הקובץ (באסקי) ומספר ה-I-NODE שלו. כל שאר המידע לגבי סוג הקובץ, גודלו, זמן ותאריך, בעל הקובץ והבלוקים בדיסק נמצאים בטבלת ה-I-NODE.
כאשר פותחים קובץ, מערכת הקבצים משתמשת בשם הקובץ כדי לאתר את הבלוק שלו, בדיסק. נעקוב אחר החיפוש של המסלול /user/ast/mbox. אנו נשתמש ב- UNIX כדוגמא, אולם האלגוריתם בעקרון זהה בכל מערכת היררכית של ספריות. תחילה מערכת הקבצים מאתרת את ספרית השורש. ב-I-NODE UNIX של השורש ממוקם במקום קבוע בדיסק.
אחר, המערכת בודקת את הרכיב הראשון במסלול, user, בספרית השורש, כדי למצוא את מספר I-NODE של קובץ user/. איתור I-NODE לפי מספרו הוא מיידי, היות ולכל אחד יש מקום קבוע בדיסק. מה-I-NODE המערכת מאתרת את הספרייה עבור user/ ומחפשת את הרכיב הבא, ast, בתוכו. עם מציאת הכניסה של ast, יש לו I-NODE עבור ספרית user/ast/. מה-I-NODE ניתן למצוא את הספרייה עצמה, ושם לחפש את mbox. ה-I-NODE עבור הקובץ הזה נקרא את הזיכרון ונשמר שם עד לסגירת הקובץ. תהליך החיפוש מודגם באיור 4-16.
חיפוש במסלול יחסי זהה לחיפוש במסלול אבסולוטי, אלא שהוא מתחיל בספרית העבודה, במקום בספרית השורש. לכל ספריה ישנן כניסות עבור .
ו- .. שנמצאות שם עם הווצרות הספרייה. בכניסה . יש את מספר I-NODE של הספרייה הנוכחית, ובכניסה של .. יש את מספר I-NODE של ספרית ההורה. לפיכך, הפרוצדורה המחפשת /dick/prog.c.. תחפש בפשטות .. בספרית העבודה, תמצא את מספר I-NODE עבור ספרית ההורה, ואזי תחפש את הספריה dick. אין צורך במכניזם מיוחד לטיפול בשם זה. עבור מערכת הספריות זוהי מחרוזת אסקי רגילה.
ב-UNIX גם למדריך יש מנגנוני הרשאה לגישה.

עמוד 167

4.3.3 שיתוף קבצים בין משתמשים שונים
כאשר כמה משתמשים עובדים יחד על פרויקט, לעיתים קרובות הם צריכים לחלוק בקבצים משותפים. לפיכך, נוח לקובץ המשותף להופיע במקביל בספריות שונות השייכות למשתמשים שונים. איור 4-17 מראה זאת.
LINK – קישור בין ספריה מסוימת וקובץ שנמצא בספריה אחרת. הקישור אפשרי רק ממדריך לקובץ, אך לא הפוך. לאחר LINK מערכת הקבצים הופכת להיות DIRECTED ACYCLIC GRAPH או DAG במקום להיות עץ.
הבעיות שמתעוררות עקב השיתוף:
נתחיל במקרה שכל הספריות ממש מכילות את הכתובות של הבלוקים בדיסק, בהם מאוחסן הקובץ (כמו ב-CP/M). אזי כאשר הקובץ יקושר, תועתק כתובתו בדיסק עבור הספרייה המעונינת בקובץ (השייך בספריה אחרת). ואז במקרה של הוספה (הגדלת הקובץ) הבלוק החדש ירשם רק בכניסה של הספרייה שעשתה זאת, כך שבמקום האחר השינוי בקובץ לא יבוא לידי ביטוי.

עמוד 169

ניתן לפתור בעיה זו בשתי דרכים:

1. יש להחזיק טבלה קטנה השייכת לקובץ ובה כל כתובות UNIX של הבלוקים בהם מאוחסן הקובץ (כמו I-NODE).

2. יצירת קובץ בספריה שרוצה שיתוף עם קובץ אחר בטיפוס LINK שבו מצוין רק המסלול למיקומו הפיסי של הקובץ המשותף, ולכן כאשר קוראים מתוך קובץ מטיפוס LINK, למעשה ניגשים לקובץ הפיסי. גישה זו קרויה קישור סימלי – SYMBOLIC LINKINK. הקישור הסמלי מאפשר ביזור מלא של המערכת מכיוון שהוא תלוי רק במחרוזת אסקי היכולה להיות כתובה בקובץ כלשהו בכל רחבי הרשת. קישור רגיל תלוי במידע על I-NODE מסוים המתאים לקובץ, והוא בהכרח ייחודי למערכת מסוימת.

הבעיה מתעוררת כאשר הספרייה המקורית, בעלת הקובץ רוצה למחקו, ולאפס את I-NODE, הספריות המקושרות ל-LINK יצביעו עתה ל-I-NODE שלא נמצא בתוקף. אם I-NODE ישמש מאוחר יותר עבור קובץ אחר, הספריות המקושרות אליו ב-LINK יצביעו על קובץ שגוי. המערכת יכולה לראות מספירת הכניסות לקובץ ב-I-NODE של הקובץ, שהוא עדיין נמצא בשימוש, אולם אין היא יכולה לדעת מיהן כל הספריות המצביעות על הקובץ, על מנת למחוק את ההצבעה. לא ניתן לאחסן מצביעים לספריות ב-I-NODE היות ומספר הספריות לא מוגבל.

הפתרון הוא לרשום את בעל הקובץ, בתוספת של מונה מספר הספריות המצביעות על הקובץ. כאשר מתבצע שיתוף מועלה המונה. כאשר הספרייה שבבעלותה הקובץ רוצה להסירו, היא תישאר בעלת הקובץ אך מונה הספריות יעודכן (יקטן ב- 1). הקובץ יימחק לגמרי כאשר כל אחת מהספריות המשותפות תסיים לעבוד עם הקובץ המשותף, המונה יקטן ב- 1. הספרייה תמחק כאשר המונה יהיה שווה ל- 0.

בקישור סימבולי בעיית מחיקת קובץ אינה קיימת מכיוון שרק לבעל הקובץ יש הצבעה ישירה לטבלת ה- I-NODE. לשאר הספריות יש רק את שם המסלול אליו, ולכן כאשר מוחקים קובץ תמחק ההצבעה אליו וכל ניסיון לגשת דרך הספריות האחרות יניב הודעה שהקובץ אינו נמצא.

עמוד 170

חיסרון קישור סימבולי היא: יש לקרוא את הקובץ שמכיל את המסלול, אחר כך לנתח את המסלול רכיב אחר רכיב עד להשגת I-NODE הדרוש. פעולות אלו דורשות גישות רבות לדיסק, וזמן רב. יתר על כן, יש צורך במקום נוסף לאחסון המסלול של הקובץ לכל ספריה שרוצה לשתף את הקובץ.

יתרון הקישור הסימבולי שהוא מאפשר ביזור מלא של מערכת הקבצים ושיתוף קבצים במערכות שונות ברחבי העולם המקושרות ברשת.

בעיה נוספת בקישור (רגיל או סמלי): לקבצים שיתופיים יש מספר מסלולים ולכן נוצרות כפילויות, לדוגמא בתוכנית גיבוי.

4.3.4 ניהול שטח הדיסק

תחילה נתאר כמה מהיבטים הטכניים הקשורים לדיסק:

1. הדיסק מורכב מכמה משטחים שכל אחד מהם דומה לתקליט. לכל צד של כל משטח מוצמד ראש קורא\כותב. בטכנולוגיה הנוכחית, כל הראשים נעים במקביל – אין אפשרות שראשים שונים יהיו בו זמנית אל אופנים (CYLINDERS) שונים. אם כי אפשרות כזו הייתה עוזרת למשל בשעת העתקה ממקום למקום בדיסק, כאשר המסלול הנקרא נמצא על אופן שונה מזה של המסלול הנכתב. במקרה זה היה נוח למקם שני ראשים במקומות בלתי תלויים זה בזה. הצורך לטייל בין האופן של המסלול הנקרא לבין האופן של המסלול הנכתב גוזל זמן רב.
2. כל אחד מהמשטחים מחולק למספר קבוע של מסלולים מקבילים (TRACKS).
3. קבוצה של מסלולים מקבילים בכל המשטחים נקראת אופן (CYLINDER). הראשים נמצאים תמיד במקביל על גבי המסלולים של אותו האופן.
4. כל מסלול מחולק לגזרות (SECTORS).

ברגע שידועה לנו הכתובת הפיסית של נתונים על הדיסק, צריכים להתבצע עוד שני דברים כדי שנוכל לקרוא או לכתוב את הנתונים:

א. הראש צריך להגיע למסלול המתאים. זהו זמן החיפוש (SEEK TIME).

ב. הדיסק צריך להסתובב כדי שהראש יגיע לגזרה המתאימה. זהו זמן הסיבוב (ROTATION TIME).

ברור כי לשיטה של חלוקת הדיסק ליחידות של בלוקים והרכבת הקבצים מבלוקים יש יתרונות גדולים הן מבחינת ניצול השטח והן מבחינת החיסכון בזמן, שכן אין צורך להזיז את הקובץ כולו ממקום למקום כאשר השטח הרצוף אוזל.

החיסרון המרכזי של שיטה זו הוא שכאשר הקובץ תופס שטח רצוף על הדיסק, קריאתו מהירה בהרבה. במשך הזמן, קבצים ארוכים הופכים לשרשרות של בלוקים מבודדים, דבר המאט את הגישה לקובץ. בעיה זו אפשר לפתור באמצעות תהליכים מיוחדים הפועלים בזמן עומס נמוך ומעבירים את הקבצים לבלוקים רצופים ככל האפשר.

ישנן באופן כללי שתי אסטרטגיות לאחסון N בתים בדיסק: איחוד N בתים פנויים שיש בדיסק, אחסון ברצף, או הפרדת הקובץ למספר בלוקים נפרדים בדיסק.

לאחסון ברצף של בתים יש חסרון בכך שאם הקובץ יגדל יהיה צורך להזיזו בדיסק והדבר גוזל זמן ומשאבים. היתרון הוא שהגישה והקריאה מהירה יותר.

מסיבה זו רוב מערכות הקבצים משתמשות בחלוקת קבצים לבלוקים שאין צורך להזיזם. כך מתקבל ניצול שטח טוב יותר, וחסכון בזמן עקב אי תזוזה.

<u>גודל הבלוק</u>

באופן טבעי רצוי שגודל הבלוק יהיה כפולה שלמה של אחד הגדלים המוכתבים על ידי הנתונים הפיסיים של הדיסק כגון גזרה, מסלול, אופן וכו'. זאת, על מנת לחסוך בתנועות הזרוע בעת קריאה או כתיבה של בלוק שלם.

במידה והבלוק גדול מידי, אזי יהיה בזבוז שטחים גדול, בגלל קבצים קטנים התופסים חלק קטן בלבד מהבלוק. מאידך, אם הבלוק קטן מידי, אזי כל קובץ ימוקם על בלוקים רבים. קריאת כל בלוק דורשת זמן חיפוש, וזמן סיבוב, לכל בלוק, כך שקריאת קובץ המורכב מהרבה בלוקים תהיה איטית.

לכן מסתבר כי מבחינת ניצול השטח, גודל הבלוק האופטימלי מושפע בעיקר מגודלו השכיח של קובץ במערכת. כדי להגיע לניצול שטח יעיל, גודלו של הבלוק חייב להיות לכל הפחות גודלו של הקובץ השכיח.

<u>עמוד 171</u>

<u>מעקב אחרי השטח החופשי בדיסק</u>

ישנן שתי שיטות עיקריות למעקב אחר השטח החופשי בדיסק:

א. אחזקה של מספרי הבלוקים הפנויים ברשימה מקושרת. הרשימה בנויה מבלוקים כשכל בלוק מאחסן מקסימום של מספר בלוקים פנויים. עם בלוקים בני 1K, ומספרי בלוקים בני 16 ביט, כל בלוק ברשימת הפנויים יכיל את מספרי 511 בלוקים פנויים. דיסק בן 20M זקוק עבור רשימת הבלוקים פנויים, למקסימום 40 בלוקים כדי להכיל את כל 20K מספרי הבלוקים.

ב. מפת סיביות – BIT MAP – דיסק בעל N בלוקים דורש מפת סיביות בעלת N ביטים, כאשר בלוק פנוי מסומן ב- 1, ובלוק מוקצה מסומן ב- 0. דיסק בגודל 20M דורש 20K ביטים עבור המפה, מה שדורש 3 בלוקים בלבד. כלומר מפת הסיביות דורשת פחות מקום מאשר הרשימה המקושרת (אלא אם כן הדיסק ממש מלא). אם יש מספיק זיכרון לאחסן את מפת הסיביות אזי שיטה זו באופן כללי עדיפה.

<u>עמוד 172</u>

<u>הגבלת השימוש בדיסק</u>

מנהל המערכת מקצים לכל משתמש מקסימום שטח דיסק - בלוקים המותרים לשימוש על מנת שהשטח הפנוי בדיסק לא יתמלא במהרה. הטכניקה שמשתמשים בה היא כזו שכאשר משתמש פותח קובץ, התכונות, והכתובות בדיסק נרשמות בטבלת הקובץ בזיכרון הראשי. בין התכונות ישנה הכניסה האומרת מיהו בעל הקובץ. כל הגדלה של הקובץ נרשמת ונבדקת למול סה"כ השטח שמוקצה למשתמש. קיימת טבלה נוספת המכילה את המכסה של כל משתמש, עם הקובץ הפתוח, אפילו אם הקובץ נפתח על ידי מישהו אחר. הטבלה באיור 4-21.

בכל כניסה של המשתמש למערכת מתבצעת בדיקה מה מצב השטח העומד לרשותו, ובהתאם מותרת כניסתו למערכת.

ישנם שני סוגי גבולות SOFT LIMIT, שאחריו מתחילות להופיע אזהרות בנדון, ו-HARD LIMIT , כאשר כל ניסיון הוספה לקובץ במצב HARD LIMIT יגרור תוצאת ERROR. מחיר השיטה – פגיעה בפרטיות.

עמוד 173

4.3.5 אמינות מערכת הקבצים

הרס מערכת הקבצים הוא לעיתים קרובות אסון גדול יותר מהרס המחשב. כאשר מחשב נהרס, ניתן להחליפו בעלות כספית כלשהי. אולם כאשר מערכת הקבצים אובדת, אם מסיבה פיסית, או מבעית תוכנה, אזי קשה לשחזר את המידע . בעוד מערכת הקבצים אינה יכולה להציע כל הגנה נגד הרס פיזי של החומרה (כגון דיסק), היא יכולה לעזור להגן על המידע.

עמוד 174

להלן מספר טכניקות המגבירות את אמינות מערכת הקבצים:

בלוקים מקולקלים

דיסקטים בדרך כלל מפתחים בלוקים מקולקלים עם זמן השימוש בהם, לדיסקים בדרך כלל יש בלוקים מקולקלים כבר מההתחלה.

ישנם שני פתרונות לבלוקים מקולקלים:

א. פתרון חומרה – להקצות סקטור בדיסק לבלוקים מקולקלים. כאשר הבקר מאותחל הוא סורק ראשית את הבלוקים המקולקלים ושם בלוק פנוי שיחליף אותם, וממפה בהתאם.

ב. פתרון תוכנה – דורש מהמשתמש או מערכת הקבצים ליצור בזהירות קובץ שיכיל את כל הבלוקים המקולקלים. טכניקה זו מורידה אותם מרשימת הבלוקים הפנויים כך שנתונים לא יוכלו לשבת שם. יש לשים לב לא לכתוב או לקרוא נתונים לקובץ זה (פתרון מערכת ההפעלה).

הבלוקים החשובים ביותר בכל דיסק הם אלה המכילים מידע על מערכת הקבצים בדיסק. אם מידע זה אובד כל מערכת הקבצים קורסת.

גיבוי מערכת הקבצים

מלבד הגיבוי באמצעים חיצוניים כגון דיסקטים וקלטות מגנטיות קיימת שיטה שבה מחזיקים שני דיסקים במחשב. שיטה זו קלה ליישום ושיחזור הנתונים אך מבזבזת חצי ממקום האחסון מכיוון שכל דיסק מחולק לשטח עבור נתונים ושטח עבור גיבוי (שרטוט 4-22). הנתונים של דיסק 0 מגובים לתוך דיסק 1 ולהפך, כך שאפילו אם אחד הדיסקים נהרס לחלוטין כל החומר נשמר.

לגיבוי אוטומטי באמצעות דיסק יש יתרון נוסף – המהירות שבה ניתן לשחזר את המידע. כאשר הגיבוי נעשה באמצעות סרטים מגנטיים, השחזור דורש בדרך כלל התערבות ידנית ולפיכך הוא איטי בהרבה. השימוש בסרטים מגנטיים נעשה אפוא במידה רבה משיקולי מחיר.

יתרון גיבוי על סרט מגנטי הוא שניתן לאחסנו בנפרד, ובמקרה של שריפה הנתונים נשמרים, להבדיל מגיבוי על דיסק.

שיטה חלופית לביצוע יומיומי זה נקראת INCREMENTAL DUMPS. בשיטה זו מבצעים גיבוי מלא רק אחת לפרק זמן מסוים – שבוע, חודש, וגיבוי יומי רק לקבצים ששונו מאז הגיבוי האחרון (המלא), ובגרסה טובה יותר מאז גיבוי כלשהו אחרון.

MS-DOS מספקת עזרה בנושא הגיבוי. לכל קובץ יש ביט הנקרא ARCHIVE BIT. כאשר הקבצים מגובים שדה זה מאותחל ל- 0, וכאשר קובץ משתנה שדה זה הופך ל- 1. בעת הגיבוי, תוכנית הגיבוי בודקת אם הביט דלוק, ומגבה רק קבצים אלו.

עמוד 175

עקביות מערכת הקבצים

אם מערכת קורסת לפני שקבצים ששונו, נכתבים בדיסק, מערכת הקבצים יכולה להישאר במצב לא עקבי בייחוד אם הבלוקים שלא נכתבו שייכים לטבלת ה-I-NODE , לספריות או לרשימות של הבלוקים הפנויים.

על מנת להתמודד עם בעיית עקביות מערכת הקבצים, לרוב המחשבים יש תוכנית עזר שבודקת אם עקביות מערכת הקבצים. ניתן להריץ אותן כאשר המערכת עולה, ובמיוחד אחר תקלה או נפילה. המחיר בהחזרת המערכת למצב עקבי יכול להיות גבוה ביותר עד

כדי מחיקת קבצים. לכן תוכניות כמו CHKDSK ב- MS-DOS או FSCK ב-UNIX, מטרתם היא לשחזר את עקביות מערכת הקבצים תוך גרימת נזקים מינימליים.
יש שתי בדיקות עקביות שנעשות:

1. בדיקה לבלוקים – התוכנית בונה טבלה עם שני מונים לכל בלוק. שניהם מאותחלים ל- 0. המונה הראשון עוקב כמה פעמים הבלוק מופיע בקובץ.
המונה השני מקליט באיזה תכיפות הוא מופיע ברשימת הפנויים (או מפת ביטים של הבלוקים הפנויים).

התוכנית קוראת את כל ה- I-NODE. כאשר מתחילים ב- I-NODE, ניתן לבנות רשימה של כל מספרי הבלוקים שהשתמשו בהם בקבצים.
כאשר כל בלוק נקרא, הוא מתווסף למונה של הטבלה הראשונה. התוכנית אז בודקת את רשימת הבלוקים הפנויים שבמפת הביטים, כדי למצוא את כל הבלוקים שלא בשימוש.
בכל מקרה שמוצאים בלוק ברשימת הפנויים הוא מתווסף למונה בטבלה השניה.
(מחזיקים שתי טבלאות. טבלה אחת לבלוקים בשימוש, וטבלה שניה לבלוקים פנויים. כאשר המערכת עקבית, מצב הסיביות (0,1) צריך להיות הפוך בין הטבלאות.)

עמוד 176

אם יש עקביות במערכת הקבצים לכל בלוק יהיה 1. או בטבלה הראשונה, או בטבלה השניה, ראה שרטוט 23-4. למרות זאת כתוצאה מהתרסקות יכולות להתעורר כמה בעיות:

בלוק חסר – בלוק שאינו מופיע ברשימת הבלוקים הפנויים, אך אינו מוקצה לאף קובץ. בלוקים חסרים לא גורמים נזק אולם הם מבזבזים מקום, ומפחיתים את תכולת הדיסק. הפתרון לבלוקים חסרים הוא להוסיפם לרשימת הבלוקים הפנויים.
בלוק חופשי מיותר - מצב נוסף שיכול לקרות הוא בלוק המופיע ברשימת הבלוקים החופשיים יותר מפעם אחת. בטבלת הבלוקים הפנויים המונה = 2 ולא 1 כפי שצריך להיות (הדבר יכול להתרחש רק במצב שהטבלה רשימה מקושרת ולא במפת סיביות). הפתרון הוא פשוט לעדכן את המספר ל- 1.

בלוק עם הקצאה עודפת – בלוק המוקצה ליותר מקובץ אחד. (המונה בטבלת בלוקים בשימוש גדול מ-1 בבלוק מסוים). המצב הגרוע ביותר שיכול לקרות. למשל אם אחד מקבצים אלו יפונה, אזי יהיה מצב שאותו בלוק יהיה ברשימת הפנויים, וברשימת הבלוקים שבשימוש, באותו זמן. אם שני הקבצים יתפנו אזי הבלוק ירשם ברשימת הפנויים, פעמיים. הפתרון הוא לקחת בלוק פנוי, להעתיק את תוך הבלוק הבעייתי אליו, ולשייכו לאחד מהקבצים.

בלוק חופשי ומוקצה – בלוק שמופיע הן ברשימת הבלוקים הפנויים והן ברשימת הבלוקים המוקצים. אם הבלוק משויך רק לקובץ אחד, אולם מופיע בשתי הרשימות אזי הפתרון פשוט, יש להסירו מרשימת הפנויים.
בדיקה למערכת הספריות:
מחזיקים טבלה של מונים, לכל קובץ, במקום לכל בלוק. מתחילים בספרית השורש, ובאופן רקורסיבי יורדים במורד העץ, ובודקים כל ספריה במערכת הקבצים. עבור כל קובץ בכל ספריה, מגדילים את המונה בטבלת ה- I-NODE. כך בסופו של התהליך, כל טבלת I-NODE של קובץ מחזיקה את מספר הספריות שמשתמשות בקובץ. מספר זה מושווה למונה הקבצים המשותפים המאוחסן בטבלת ה- I-NODE . במערכת קבצים עקבית, שתי המונים יהיו שווים בערכם.

עמוד 177

אם מונה ה-LINK אשר ב- I-NODE גבוה מזה בספריות אזי אפילו אם כל הקבצים יוסרו מהספריות, מונה ה-LINK לא יהיה שווה ל- 0, וה- I-NODEלא יוסר. הבעיה לא חמורה אולם הדבר מהווה בזבוז מקום על הדיסק עם קבצים אשר לא נמצאים באף ספריה. יש לתקן את הבעיה על ידי תיקון מונה LINK שב-I-NODE, לערכו הנכון. עם הערך הנכון הוא 0 אזי יש למחקו.

הבעיה השניה היא קטסטרופה, בפוטנציה. אם שתי ספריות מקושרות לקובץ, אולם ה--I NODE חושב שהן אחת, אזי כאשר אחת מהספריות תוסר, המונה ב-I-NODE יתאפס, ובעקבות זאת מערכת הקבצים תסמנו שלא בשימוש ותשחרר את הבלוקים שהיו קשורים

אליו. פעולה זו תגרום לאחת מהספריות להצביע ל-I-NODE שלא בשימוש, אשר הבלוקים שלו יוקצו עוד מעט לקבצים אחרים. הפתרון הוא לאלץ את המונה של הקישור להיות שווה למונה שנמצא ב-I-NODE הספריות.
שתי פעולות אלו, בדיקת הבלוקים, ובדיקת הספריות, לעיתים קרובות מאוחדות, משיקולי יעילות.
ישנן בדיקות נוספות אפשריות. לדוגמא, לספריות יש פורמט מוגדר, עם מספרי I-NODE ושמות באסקי. אם מספר I-NODE גדול ממספרי I-NODE שעל הדיסק, הספרייה תינזק.
במערכת ההפעלה MS-DOS ומערכות הפעלה אחרות, מגנים על המשתמש מפני עצמו כאשר הוא מוחק קבצים. כאשר קובץ מוסר מה שמתרחש הוא שביט מסוים מאותחל בכניסה בספריה, או בטבלת ה-I-NODE המסמן שהקובץ הוסר. הבלוקים בדיסק אינם חוזרים לרשימת הבלוקים הפנויים עד אשר אכן צריכים אותם. אם המשתמש מזהה את הטעות שלו מיידית, זה אפשרי להריץ תוכנית שתחזיר לאחור, כגון UNDELETE.
ב-UNIX הדבר אינו אפשרי מכיוון שבעת שקבצים נמחקים, הבלוקים שהיו קשורים אליהם משוחררים וחוזרים מיידית לרשימת הבלוקים הפנויים.

עמוד 178

4.3.6 ביצועי מערכת הקבצים

בשל הפער העצום בין מהירות הגישה לדיסק לבין מהירות עיבוד הנתונים בזיכרון הראשי, הגישה לדיסק יכולה להפוך בנקל לצוואר הבקבוק של המערכת כולה. הדיון כאן מתמקד בדרכים להתגבר על מכשלה זו, וברור כי הדבר הראשון שיש לעשות הוא לצמצם ככל האפשר את מספר הגישות לדיסק.
הטכניקה השכיחה ביותר להפחית את הגישות לדיסק היא מנגנון ה- BLOC CACHE או BUFFER CACHE - זיכרון מטמון. זיכרון מטמון הוא אוסף של בלוקים השייך לוגית לדיסק אך נשמר בזיכרון מסיבות של ביצוע.
ישנם כמה אלגוריתמים לניהול זיכרון מטמון, אחד מהנפוצים הוא אחזקת הבלוקים האחרונים שנקראו בו כך שבכל בקשת קריאה בודקים ראשית אם הבלוק נמצא בזיכרון מטמון ואם לא נגשים לדיסק, קוראים את הבלוק אל זיכרון המטמון, ואחר כך מעתיקים אותו לכל דורש. עבור כל הכנסת בלוק חדש יש להוציא בלוקים ישנים, וניתן להשתמש בכל אחד מהאלגוריתמים שתוארו במנגנון הדפדוף, כמו FIFO, LRU.
ההבדל בין LRU בדפדוף לבין LRU בזיכרון מטמון הוא שפניות זיכרון מטמון הן יחסית יותר נדירות ולכן אפשרי להחזיק את כל הבלוקים ברשימה מקושרת בעלת סדר מדויק, קבוע. אך בעובדה זו יש בעיה שהופכת את LRU לאלגוריתם בלתי רצוי – אם בלוק חיוני כגון I-NODE נקרא לזיכרון המטמון, ושונה אך לא נרשם לדיסק, בעת נפילה המערכת תישאר לא עקבית. אם הבלוק נמצא בסוף הרשימה, ייקח זמן עד שהוא יגיע לראש הרשימה וייכתב בדיסק.
בניה נוספת היא שיש בלוקים הנקראים לתוך הזיכרון מטמון אך לעיתים רחוקות צריכים אותם פעמיים, בפרק זמן קצר. בעיות אלו הובילו לשיפור LRU, תוך שימת לב לשני גורמים:

1. האם יצטרכו את הבלוק בקרוב?
2. האם הבלוק חיוני לעקביות המערכת?

עבור שתי השאלות הבלוקים מחולקים לקטגוריות כמו בלוקים ל-I-NODE, בלוקים לספריות, בלוקים מלאים נתונים, בלוקים מלאים חלקית בנתונים. בלוקים שלא יזדקקו להם שוב יהיו בתחילת הרשימה. בלוקים שיזדקקו להם שוב, כמו בלוקים מלאים חלקית שכותבים בהם, יהיו בראש הרשימה, כדי שיישארו בזיכרון זמן ארוך יותר.
השאלה השניה לא תלויה בראשונה. אם בלוק חיוני לעקביות מערכת הקבצים (באופן בסיסי הכל, מלבד בלוקים של נתונים), והוא עבר שינוי, יש לכתבו לדיסק מייד, ללא קשר למקומו ברשימת LRU. על ידי כתיבת בלוקים חיוניים, באופן מיידי, מפחיתים באופן משמעותי את האפשרות לנזק למערכת הקבצים.

עמוד 179

בלתי-רצוי להחזיק בלוקים של נתונים בזיכרון מטמון לפני כתיבתם באמצעי אחסון (כגון דיסק). אפילו אם המשתמש מבצע שמירת נתונים אחת לכמה זמן, עדיין ייתכן שהנתונים בזיכרון מטמון, ואם תקרה נפילה, הנתונים לא ישמרו.
מערכות נוקטות בשתי גישות להתמודד עם מצב זה:

UNIX - ישנה קריאת מערכת SYNC שמאלצת את כל הבלוקים ששונו להיכתב מיידית בדיסק. כאשר המחשב מאותחל, יושבת ברקע תוכנית הנקראת בדרך כלל update שמריצה לולאה אינסופית של קריאת המערכת SYNC הישנה 30 שניות בין קריאה לקריאה – בעקבות כך, אם תקרה נפילה רק הנתונים שעודכנו ב- 30 שניות האחרונות ילכו לאיבוד.

MS-DOS – כל בלוק ששונה נכתב מיידית לדיסק. מערכות כאלה נקראות WRITE-THROUGH CACHS. גישה זו דורשת יותר גישות לדיסק על כל תו ושורה אך בעת נפילה של דיסק אף נתון לא הולך לאיבוד. בעוד ב-UNIX הדבר תלוי מתי התבצעה קריאת SYNC. ההבדל הגישות נובע מכך ש- UNIX פותחה בסביבה שכל הדיסקים היו קשיחים ואילו MS-DOS החלה את דרכה עם דיסקטים. כאשר הדיסקים הם קשיחים, גישת UNIX יעילה יותר.

זיכרון מטמון אינו הדרך היחידה לשפר את ביצועי מערכת הקבצים. לאחר שמיצינו את כל האפשרויות שבזיכרון המטמון, ואנו נאלצים לבקר בדיסק, רצוי כי ביקורים אלו יהיו קצרים ונדירים ככל האפשר. וכדי שהביקורים אכן יהיו קצרים יש לחסוך ככל האפשר בתנועת הזרוע של הדיסק ובהמתנה לסיבובו.
להלן כמה שיטות לקיצור זמן הגישה לנתונים:

א. הקצאת בלוקים קרובים ככל האפשר זה לזה – בלוקים אשר סביר שהגישה אליהם תהיה סדרתית, עדיף שיהיו באותו אופן. כאשר קובץ פלט נכתב, מערכת הקבצים מקצה בלוקים, אחד בכל פעם, לפי הדרישה. אם בלוקים פנויים נרשמו במפת ביטים, וכל מפת הביטים נמצאת בזיכרון הראשי, קל לבחור בלוקים פנויים קרובים אחד אל השני. כמו כן ניתן לבנות את מפת הסיביות כך שתייצג בקלות גם את זמן המעבר בין הבלוקים. כל שנדרש הוא שבלוקים הנמצאים על אותו אופן, ייוצגו על ידי סדרות סיביות רצופות. בכל פעם שנזדקק לבלוק חדש עבור קובץ, נחפש בלוק פנוי המיוצג על ידי הסיבית הקרובה ביותר במפת הסיביות, לסיבית הקודמת ששימשה אותנו. כך קרוב לודאי ששני הבלוקים העוקבים יימצאו על אותו אופן. בירור המרחק היחסי של בלוקים הנמצאים ברשימה משורשרת היא משימה מורכבת יותר – אם כי אפשרית. מכל מקום הזמן הנדרש להשוואה זו גדול הרבה יותר כאשר הבלוקים הפנויים משורשרים. אם רשימת הבלוקים הפנויים היא רשימה מקושרת, שחלקה יושב בדיסק, קשה יותר להקצות בלוקים קרובים. אולם למרות זאת ניתן לנסות להקצות בלוקים קרובים. כאשר הגישה היא לשמור את הבלוקים ביחידות של כמה בלוקים.

עמוד 180

ב. הקצאה ביחידות הכוללות מספר בלוקים - מטרתה להפחית את זמן הסיבוב. כאשר מקצים בלוקים, המערכת מנסה להקצות בלוקים צמודים, הנמצאים באותו אופן, לקובץ. החיסרון של שיטה זו הוא בזבוז מקום.

ג. מיקום טבלאות המערכת הקשורות לטיפול הקבצים - צוואר בקבוק נוסף במערכת שמשתמשת ב-I-NODS הוא שקריאה, אפילו מקובץ קצר דורשת שתי כניסות לדיסק. אחת עבור I-NODE והשניה עבור הבלוק. איור 24-4 מראה את מיקום I-NODE. כאן כל הטבלאות או I-NODE נמצאים קרוב לתחילת הדיסק, כך שהמרחק הממוצע מ-I-NODE אל הבלוקים שלו הוא בערך חצי ממספר האופנים, דבר הדורש זמן חיפוש ארוך. אפשר למקם את הטבלאות באמצע הדיסק, במקום בהתחלה. דבר זה משפר את החיפוש הממוצע פי שתיים. רעיות אחר הוא לחלק את הדיסק, לקבוצות אופנים, בכל אחד יש I-NODE, הבלוקים שלהם ורשימת פנויים. כאשר נוצר קובץ חדש, ניתן לבחור כל I-NODE, ואז המערכת מנסה למצוא בלוקים באותה קבוצת אופן. אם כל הבלוקים תפוסים, אזי נחפש בלוק בקבוצת אופנים קרובה. הרעיון הוא שרצוי למקם

את הטבלאות (כגון I-NODE) קרוב ככל האפשר לקבצים שהן מייצגות. היות ולא ניתן לגשת לקובץ לפני איתורו בטבלת הקבצים, ולכן צפוי כי הראש יעבור פעמים רבות בין המסלול של טבלת הקבצים לבין מסלולי הקבצים הרשומים בטבלה זו.

4.4 אבטחה

מערכת קבצים לעיתים קרובות מכילה מידע בעל ערך גבוה עבור המשתמש. הגנת המידע בפני חדירה לא חוקית היא בעיה חשובה בכל מערכת קבצים.

עמוד 181

4.4.1 סביבת האבטחה

משתמשים במילה "אבטחה" כאשר מתייחסים לבעיה הכללית, ובמושג "מנגנון הגנה" כאשר מתייחסים למנגנון עצמו של מערכת ההפעלה ששומר על האינפורמציה במחשב. לאבטחה יש כמה צדדים:

א. איבוד נתונים – ישנם כמה סיבות לכך:

1. אסונות טבע כגון שריפות, רעידות אדמה, מלחמות, מהומות, ועוד.
2. תקלות חומרה\תוכנה.
3. טעויות אנוש.

ניתן להתמודד עם רוב הבעיות על ידי שמירה במקום מוגן גיבויים תדירים ככל האפשר.

ב. מתנכלים\פולשים למערכת - ישנם שני סוגים. פולשים פסיביים שרק רוצים לקרוא נתונים שלא שייכים להם. פולשים אקטיביים שרוצים לשנות ולשבש נתונים. כאשר מאבטחים מערכת יש לזכור כנגד אלו פולשים יש להגן על המערכת. להלן כמה קטגוריות נפוצות:

1. פולש מיקרי – לאנשים רבים יש מסופים במערכת שיתוף בזמנים על שולחנם, וטבע האדם הוא כזה שכמה מהם יקראו דואר אלקטרוני או קבצים של אחרים, במידה ואין מחסומים בדרך. לרוב מערכות UNIX יש ברירת מחדל שניתן לקרוא בכל הקבצים.
2. ריגול פנימי – סטודנטים, מתכנתים, ואנשים טכניים אחרים, מחשיבים כאתגר אישי, לפצח את אבטחת מערכת המחשבים המקומית. לעיתים קרובות הם מיומנים מאוד, ומוכנים להשקיע זמן רב בנושא.
3. ניסיון עיקש להרוויח כסף – למשל ניסיון לחדור למערכת בנק, ולגנוב מהבנק.

עמוד 182

או לחלופין לאיים בהריסת המערכת באם לא ישולם להם כופר.

4. ריגול מסחרי או בטחוני.

ההשקעה הכספית הדרושה לאבטחת המערכת תלויה כנגד מי מהפולשים יש לאבטח את המערכת.

פן נוסף של בטיחות הוא – פרטיות וזכויות הפרט – הגנת הפרט כנגד שימוש לרעה במידע עליו. זוהי בעיה לגלית ומוסרית. מה מידת ההרשאה שיש לאפשר לרשויות? האם הם יכולים להסתכל בנתונים פרטיים של האזרח (אפילו בנושאי פשע וכו').

4.4.2 פרצות בטיחות מפורסמות

הבנת התיאור של הפרצות בגרסאות הקודמות של UNIX מצריכה מעט ידע טכני, וכן הכרת הפילוסופיה של מערכת זו. ב- UNIX הכל אמור להיות גלוי! ההגנה איננה מבוססת על ידיעת סודות, אלא אמורה להיות מובנית במערכת, כאשר עקרונותיה ידועים לכל. כך למשל ידוע לכל כי הקובץ /etc/passwk מכיל את שמות המשתמשים, כולל root, שהוא בעל היכולת הבלתי מוגבלת במערכת, וברור כי השתלטות על קובץ זה וכתיבה לתוכו מאפשרות השתלטות על המערכת כולה. לכאורה ניתן היה לשמור על הקובץ על ידי החבאתו וקריאתו בשם אחר. הפילוסופיה של UNIX שונה לחלוטין: מקומו של הקובץ ושמו ידועים לכל. כל אחד יכול לקרוא אותו ואפילו להדפיסו, ולמרות זאת המערכת אמורה להיות מוגנת. גם מבנהו הפנימי ומשמעות שדותיו של הקובץ – ידועים ומתועדים, ולמרות זאת ההגנה קיימת. מובן שפילוסופיה זו דורשת מנגנוני הגנה חכמים, שכן לא ניתן להסתמך על חוסר ידיעה של הפורץ.

נעבור לרשימה קצרה של תקלות בטיחות בגרסאות קדומות של UNIX:

א. האופציה להדפיס על ידי lpr (השד המדפיס של UNIX) ואחר-כך למחוק את הקובץ, נוצלה כדי למחוק את /etc/passwd.

ב. לצורכי מציאת שגיאות יש ב-UNIX מנגנון הנקרא core dump המאפשר קריאת כל תוכן הזיכרון של תהליך ברגע נתון אל קובץ בשם core. קישור של שם זה אל קובץ ההרשאות etc/passwd/ גרם לכתיבת תוכן הזיכרון על קובץ ההרשאות.

ג. מנגנון setuid מאפשר למשתמש לקבל באופן רגעי הרשאה מיוחדת לצורך תכנית מוגדרת היטב וזאת כדי שיוכל להשתמש בחלק מההרשאות של ROOT ללא סכנה למערכת כולה, שכן התכנית מוגבלת ואיננה ניתנת לשינוי. על בסיס אפשרות זו נעשה ניסיון להשתלט על etc/passwd/ פשוט על ידי הפיכה לבעליו.

הדוגמה של MULTICS היא דוגמה לטכניקה הידועה בשם סוס טרויאני. בשיטה זו, תכנית הרסנית מתחפשת לתכנית תמימה המוכרת היטב. דוגמה מפורסמת ביותר בהקשר הזה היא החיקוי של login ב-UNIX. אחד המשתמשים משאיר על המסך את פקודות הכניסה הרגילות למערכת. משתמש אחר נופל למלכודת כאשר הוא מנסה להתחבר למערכת ומקליד אגב כך את סיסמתו, הנרשמת מיד על ידי התכנית המתחפשת.

הדוגמא של TENEX מבהירה עד כמה ידיעת התלות בין התוכנה לחומרה מסוכנת מבחינה בטיחותית. כאן הייתה למשתמש אפשרות לדעת מתי מתקיימת פסיקת דף ולהגיב עליה מיידית באמצעות פונקציה משלו. תגובה זו אפשרה פיצוח הדרגתי של הסיסמה. שם לב כי מספר הניסיונות הדרוש היה 128*n כאשר n הוא אורך הסיסמה. מספר זה קטן בהרבה מ- 128 בחזקת n שהוא מספר האפשרויות ליצירת מחרוזת אסקי באורך n.

מלבד בטיחות הסכנה שבהסתמכות על תכונות החומרה בתכנית היא מניעת אפשרות ההעברה של התכנית למכונה אחרת.

הדוגמה הבאה לקוחה ממערכת ההפעלה OS/360. וכן הסתמכו על ידיעת סדר הפעולות בזמן קריאת קובץ מוגן. האפשרות להחליף את שם הקובץ באמצעות קריאה מסרט היא כשל בטיחותי שלא הובא בחשבון.

עמוד 184

4.4.3 זחל האינטרנט

פריצת האבטחה התחילה בערב של 2 נובמבר 88 כאשר סטודנט בוגר קורנל, שיחרר תוכנית חילזון לתוך האינטרנט. פעולה זו הפילה אלפי מחשבים באוניברסיטאות, חברות ומעבדות ממשלתיות, בכל רחבי העולם, לפני שהיא אותרה וסולקה. הכל החל בויכוח שעדיין לא הוכרע. אנחנו נדון בעקרי האירוע הנ"ל. הסיפור החל ב- 88 כאשר הסטודנט גילה שני באגים במערכת של ברקלי. הם אפשרו כניסה בלתי מורשת לכל המחשבים באינטרנט. ברשת העצומה שמחברת אלפים של מחשבים בארה"ב אירופה והמזרח הרחוק. בעודו לבד הוא כתב תוכנית שמחליפה לבד ונקראת זחל.

תוכנית זו יודעת לנצל שגיאות והיא משכפלת את עצמה בכל שניה בכל מחשב, שהיא יודעת להיכנס אליו. הוא עבד על התוכנית במשך חודשים, כאשר הוא מכוונן אותה ומנסה אותה תוך כדי טשטוש עקבותיו. לא ידוע האם ההפצה שהייתה ב- 2 לנובמבר נועדה להיות מבחן או שזהו הדבר האמיתי. בכל מקרה היא הביאה את כל המערכות SUN ו- VAX לקריסה מוחלטת, ותוך שעות מרגע ההפצה. לא ידוע מה היה המניע של הסטודנט, אבל ייתכן שכל הרעיון היה מתיחת הי-טק שעקב טעות בתכנון יצאה מכלל שליטה.

באופן טכני הזחל מורכב משתי תוכניות. תיחול והזחל עצמו.

התיחול היה 99 שורות בשפת C שנקראו ll.c. היא התקמפלה ובוצעה במערכת תחת התקפה. ברגע שהיא רצה היא חיברה למחשב אליה היא מגיעה, את הזחל הראשי וביצעה אותו. אחרי שהתגברה על כמה קשיים להסתיר את קיומה, הזחל התבונן בטבלת השורש של המארח, לאבחן איזה מחשבים מחוברים אליו, ולנסות להפיץ את התיחול למחשבים האלה.

שלוש שיטות נוסו לזהם את המחשבים החדשים:

1. לנסות להריץ מעטפת הפעלה בהשתמש בפקודת rsh. חלק מהמחשבים בוטחים במחשבים אחרים ומריצים בבטחה rsh ללא אישור מיוחד. אם זה עבד מעטפת ההפעלה טוענת את תוכנית הזחל וממשיכה את הפצת הזיהום למחשבים אחרים.
2. שימוש בתוכנית קיימת בכל מערכות BSD שנקראת finger ומאפשרת לכל משתמש בכל מקום באינטרנט לכתוב finger name@site. על מנת להראות אינפורמציה על אדם במקום מסוים. האינפורמציה הזאת בדרך כלל מכילה את שמו של האדם, סיסמא, כתובת בית ועבודה, מספרי טלפון, שם של מזכירה ומספר טלפון שלה, מספר פקס. ואינפורמציה דומה. מבחינה אלקטרונית דומה לספר טלפונים.
Finger עובד בצורה הבאה: בכל אתר BSD מתרחש תהליך רקע שנקרא השד של ה-FINGER שרצה כל הזמן, ועונה על שאלות בכל האינטרנט. מה שעושה הזחל נקרא finger עם מחרוזת בת 536 ביטים כפרמטר. מחרוזת כל כך גדולה הציפה את חוצץ של השד, וכתבה מחדש את המחסניות שלו. הבאג ניצל פה את החולשה של השד לבדוק אם יש לו הצפה. כאשר השד חוזר מהפרוצדורה, זה היה בזמן שהוא קיבל את הבקשה. הוא חזר לא ל- MAIN אלא לפרוצדורה בתוך מחרוזת בת 536 ביטים במחסנית. הפרוצדורה הזאת ניסתה לבצע /bin/sh אם זה הצליח היה לזחל עכשיו מעטפת לרוץ במחשב תחת התקפה.
3. שיטה שלישית הסתמכה על באג במערכת הדואר, sendmail שמאפשרת לזחל לדוור העתק של התיחול. ברגע שהיא התבססה הזחל מנסה לפצח את הסיסמאות של המשתמש. הסטודנט לא היה צריך לחקור הרבה, כל שהיה צריך לעשות זה לשאול את אביו שהיה מומחה לביטחון בסוכנות הביטחון הלאומית של ממשלת ארה"ב, שעוסקת בפיצוח קודים, לקבל הדפס של עבודה קלאסית בנושא, שכתב אביו, עשור לפני כן עבור מעבדות בל. כל סיסמא מפוצחת אפשרה לזחל להתחבר למחשבים שלהם היו שייכות הסיסמאות. בכל פעם שהזחל הצליח לגשת למחשב חדש הוא בדק אם יש העתקים אחרים של הזחל כבר מופעלים. אם כן, ההעתק החדש יצא החוצה מלבד פעם אחת מתוך שבע שהוא המשיך להתקיים. כנראה בנסיון להמשיך להפיץ את הזחל. השימוש של אחד משבע יצר סוגים רבים מידי של זחלים, וזוהי הסיבה שכל המחשבים הנגועים נעצרו. הם זוהמו בזחלים.
אם הסטודנט היה מסתפק רק בהשתלת הזחל והפצתו, אזי אי אפשר היה לאתר אותו. הוא נלכד כאשר אחד מחבריו דיבר עם כתב לעניני מחשבים של הניו-יורק טיימס, וניסה לשכנע אותו שכל האירוע היה בטעות. שהזחל לא מזיק, ושהמפיץ מצטער. החבר לא לקח בחשבון שהסיסמא של המבצע הייתה rtm. להפוך rtm לשם הבעלים היה קל מאוד, כל שצריך היה להריץ finger. למחרת היום הסיפור היה בעמוד הראשון, והשפיע אפילו על הבחירות לנשיאות 3 שנים מאוחר יותר. הסטודנט הורשע בעבירה פדרלית, נקנס ב-10000$, 3 שנים על תנאי, ו- 400 שעות שרות לקהילה. ההוצאות המשפטיות שלו עלו על 150,000$. המשפט עורר מחלוקות רבות. הרבה בקהילת המחשבים הרגישו שהוא סטודנט מבריק, ולא מזיק, בעל עקרונות לא מזיקים שיצאו מכלל שליטה. שום דבר בזחל לא יכל להצביע על כך שהוא ניסה לגנוב או להזיק. אחרים הרגישו שהוא פושע רציני שמקומו בכלא.

עמוד 186

4.4.4 כיצד לפרוץ למערכת

הליקויים המתוארים למטה תוקנו ברובם, אבל מערכות ההפעלה הממוצעות עדיין דולפות כמו מסננת. השיטה הרגילה לבחון את בטחון המערכת היא להעסיק קבוצה של מומחים שנקראת tiger team או קבוצת חדירה. על מנת לבחון אם הם יכולים לפרוץ פנימה. חברת הברד ניסתה את אותו דבר עם קבוצת בוגרי אוניברסיטה. במהלך אותם שנים קבוצות חדירה אלה גילו מספר אזורים חלשים במערכות. כאשר דוגמים מערכת יש לבחון שהיא עומדת בהתקפות מסוג זה. להלן פרוט שיטות התקפה מוצלחות:

1. בקש דפי זיכרון, מקום פנוי בדיסק או טייפ ישנים וקרא אותם. מערכות רבות לא מוחקות נתונים ישנים לפני ההקצאה למשתמש אחר, ויש בהם אינפורמציה חשובה רבה שנכתבה על ידי הבעלים הקודמים. כל שנותר הוא פשוט לקרוא את הנתונים.

2. נסה קריאות מערכת בלתי חוקיות, או קריאות חוקיות עם פרמטרים לא חוקיים, או קריאות חוקיות עם פרמטרים חוקיים או לא הגיוניים. מערכות רבות התבלבלו בקלות.
3. התחל login ואז הקש del, rubout או break באמצע פעולת login. במערכות מסוימות תהליך בדיקת הסיסמא לא תעבוד והכניסה תתבצע בהצלחה.
4. נסה לשנות מבנים של מערכת ההפעלה השמורים במרחב המשתמש. במקרים רבים על מנת לפתוח קובץ התוכנית בונה מבנה נתונים גדול שמכיל את שם הקובץ, פרמטרים רבים אחרים שמועברים במערכת. כאשר הקובץ נקרא ונכתב המערכת לפעמים מעדכנת את המבנה בעצמה על ידי שינוי של אותם שדות אפשר לקעקע את הביטחון.
5. ניתן להתל במשתמש על ידי כתיבת תוכנית שמדפיסה "login:" על המסך ומסתלקת. משתמשים רבים ייגשו למסוף, ויקישו את הסיסמא שהתוכנית מקליטה למטרותיה הזדוניות. בדומה לטכניקת הסוס הטרויאני.
6. חפש במדריכים הכתובים היכן כתוב "אל תעשה X", נסה וריאציות רבות ככל האפשר של X.
7. לשכנע את מתכנת המערכת לשנות את המערכת כך שתדלג על מספר ביקורות בטיחות לכל משתמש שמשתמש בסיסמא שלך. התקפה זו נקראת TRAPDOOR.
8. אם כל זה נכשל, החודר יכול למצוא את המזכירה של מרכז המחשב, ולנסות להתל בה, או לשחד אותה. למזכירה יש בוודאי אינפורמציה בעלת ערך, והיא בדרך כלל מקבלת שכר נמוך, ויש לה גישה למערכת. אל תזלזל בבעיות שנגרמות על ידי צוות העובדים.

הדרך הבטוחה ביותר לשמירה על קבצים מפני קריאה בלתי רצויה היא:

א. להצפינם בשיטה מובטחת, כגון אחת משיטות המפתח הפומבי.
ב. לגבות קבצים אלו כדי למנוע פגיעה פיסית בהם.

<u>וירוסים</u>

קטגוריה מיוחדת של התקפה הם וירוסי מחשב שהפכו להיות בעיה מרכזית לרבים ממשתמשי המחשב. הוירוס הוא קטע תוכנית שמוצמד לתוכנית לגיטימית במטרה לפגוע בתוכניות אחרות. וירוס שונה מהזחל בכך שהוירוס נישא על גבי תוכנית קיימת, בעוד שהזחל הוא תוכנית מושלמת בפני עצמה.
לשניהם מטרה להפיץ את עצמם, ויכולים לגרום נזקים חמורים.
וירוס נותר גם לאחר ריבוי המערכת, זחל זה תהליך שמת בריבוי המערכת, לכן מועבר ברשת מחשבים ותלוי בחיבור זה.
וירוס טיפוסי פועל כדלקמן:

האנשים שכותבים את הוירוס, יוצרים תחילה תוכנית מועילה כגון משחקים ל-MS-DOS. התוכנית הזאת מכילה את קוד הוירוס מוחבא בתוכה. המשחק מופץ לציבור, או מחולק חינם, או נמכר במחיר צנוע על גבי דיסק גמיש. התוכנית מתפרסמת והאנשים מתחילים להעלות אותה ולהשתמש בה. לבנות וירוס זה לא קל, כך שהאנשים שעושים זאת הם בדרך כלל מבריקים, והאיכות של המשחק, או התוכנית בדרך כלל מצוינת.
כאשר מאתחלים את התוכנית היא מיד בוחנת את כל התוכניות הבינריות על הדיסק הקשיח, לראות אם הם כבר נגועים מקודם. כאשר אובחנה תוכנית לא נגועה, אזי קוד הוירוס מוצמד לסוף הקובץ, ומחליף את ההוראה הראשונה בקובץ בקפיצה לקטע הוירוס. כאשר קטע קוד הוירוס מסתיים, הוא ממשיך לבצע את ההוראה שמקודם הייתה ראשונה וממשיך בתוכנית. בדרך זו בכל פעם שהתוכנית הנגועה רצה, היא מדביקה עוד תוכניות. מלבד הדבקת תוכניות אחרות, וירוס יכול לעשות עוד פעולות כגון מחיקה, שינוי, וכו'. וירוס אחד מציג הודעת סחיטה על המסך, שמודיעה למשתמש לשלוח מזומנים, לתיבת דואר בפנמה, או להתמודד מול אובדן סופי של המידע, ונזק לתוכנה. ייתכן גם שוירוס ידביק את סקטור ה-BOOT של הדיסק, הגורם לכך שלא ניתן לאתחל את המחשב. וירוס כזה ייתכן ויבקש סיסמא, שכותב הוירוס יספק תמורת כסף.

עמוד 188

ניתן בקלות למנוע בעיות וירוסים, מאשר לתקנן. הדרך הבטוחה למנוע הדבקות בוירוס היא להכניס תוכנות רק ממקורות ידועים ובטוחים. העלאת תוכנות חינם, או שימוש בהעתקים פירטים, מדיסק גמיש הם מקור לצרות.
קיימים גם תוכניות אנטי-וירוס, חלקן עובדות פשוט על ידי איתור וירוסים מוכרים. גישה נוספת לאנטי-וירוס היא על ידי פרמוט של הדיסק הקשיח לחלוטין, כולל אזור ה-BOOT, לאחר מכן התקנה של תוכנות מוכרות ובטוחות ובדיקת גודל של כל קובץ. האלגוריתם לא חשוב, כל שדרוש שיהיה לו מספיק ביטים לפחות 16, עדיף 32. לאחר מכן, אחסון רשימה של קבצים, בזוגות במקום בטוח, או מחוץ למערכת, על גבי דיסק גמיש, או און-ליין, אבל במקום מוצפן. מתחילים בנקודה זו כאשר המערכת מאותחלת (BOOT), כל בדיקות גודל הקבצים, צריכים להיות ממוחשבים, ומושווים לרשימה המאובטחת של גודל קבצים.
במילים אחרות, האנטי-וירוס משבה את מבנה הקבצים הנוכחיים לקודמים – הדבר מביא לגילוי מוקדם, אך לא מונע הדבקות.
כל קובץ שגודלו שונה מהגודל האורגינלי מייד חשוד כנגוע. זיהום יכול להיות חמור יותר אם המדריך, במקום שיש תוכניות בינאריות, נעשה בלתי ניתן לקריאה מחדש. הטכניקה הזאת מקשה על הוירוס לשנות קבצים בינארים אחרים. אפשרות זאת ניתנת לביצוע ב-UNIX, אולם לא מתאפשרת ב-MS-DOS מכיוון שב- MS-DOS אין אפשרות להפוך את הספריות לבלתי-כתיבות בכלל, בעוד ב- UNIX ניתן. סיבה מהותית יותר היא שהוירוסים מסתמכים במידה רבה על תכונות פיסיות של המחשב, ועל גישה ישירה למערכת ההפעלה עצמה. גישה זו היא קלה ב-MS-DOS אך כמעט בלתי אפשרית ב-UNIX. UNIX אינה מאפשרת שום גישה לחומרה עצמה. כל פנייה לחומרה נעשית באמצעות קריאות המערכת בלבד.

4.4.5 עקרונות לתכנון מערכת בטוחה

וירוסים בדרך כל קורים על מערכות שולחן העבודה של המחשב. במערכות גדולות יותר נוצרות בעיות אחרות, ונחוצות שיטות אחרות לפתור אותן. מצאו מספר עיקרים שמשתמשים בהם בתכנון מערכות אבטחה:

1. תכנון המערכת חייב להיות פומבי. ההנחה שפולשים לא ידעו איך המערכת פועלת, תוליך שולל את המתכננים.
2. ברירת המחדל של המערכת צריכה להיות אין כניסה. טעויות שנגרמות כתוצאה ממניעת כניסה שהיא לגיטימית מדווחות במהירות רבה יותר מאשר איתור כניסות לא מורשות.
3. כל העת לבדוק הרשאה נוכחית. מערכות רבות בודקות הרשאה כאשר נפתח קובץ, ולא לפני כן. המשמעות היא שמשתמש שפותח קובץ, ומשאיר אותו פתוח לשבועות, ימשיך לקבל גישה, למרות שהבעלים של הקובץ מזמן שינה את ההגנה על הקובץ.
4. לתת לכל תהליך את מינימום ההרשאות האפשריות. עם לעורך בלבד יש את ההרשאה להיכנס לקובץ ולערוך אותו, אורחים עם סוסים טרוינים לא יוכלו לגרום נזק. העיקרון דומה לסכמה שמוגנת היטב, נדון בסכמות כאלה בהמשך הפרק.
5. מנגנון ההגנה צריך להיות פשוט, אחיד, ובנוי קרוב לרמה התחתונה של המערכת.
6. שיטת ההגנה צריכה להיות מקובלת מבחינה פסיכולוגית. אם המשתמשים ירגישו שההגנה על הקבצים הופכת להיות עבודה רבה מידי, הם לא יעשו זאת. למרות זאת הם יתלוננו בקול רם אם משהו ישתבש.

עמוד 189

4.4.6 זיהוי

הרבה תוכניות הגנה מבוססות על ההנחה שהמערכת יודעת לזהות כל משתמש. הבעיה של זיהוי משתמשים כאשר הם נכנסים למערכת נקראת זיהוי משתמש. רוב שיטות הזיהוי מבוססות על זיהוי משהו שידוע למשתמש.

<u>סיסמאות</u>

שיטת הזיהוי המקובלת ביותר היא לדרוש מהמשתמש להדפיס סיסמא. הגנה באמצעות סיסמא, קלה להבנה, וקלה לביצוע. שיטת הצפנת הסיסמאות ב- UNIX ידועה ופועלת באופן הבא:

המשתמש מקיש את שם המשתמש וסיסמא. הסיסמא מיד מוצפנת. תוכנית ה-LOGIN קוראת את קובץ הסיסמאות שמורכב משורות אסקי, אחד עבור כל משתמש. התוכנית עוברת שורה שורה עד שהיא מגיעה לשם המשתמש שנכנס, ומשווה את הסיסמאות – בהתאם לכך מכניסה את המשתמש או דוחה אותו.

זיהוי סיסמא קלה לפיצוח. מדי פעם אנו קוראים על קבוצות של סטודנטים שמפצחים כניסות למערכות סודיות של חברות או ממשל. את הסיסמאות ניתן לפרוץ על ידי ניסיונות חוזרים ונשנים של שם המשתמש + קומבינציה של סיסמא.

לפי מחקר על סיסמאות ב-UNIX, הם הידרו רשימה של סיסמאות שדומות אחת לשניה: שמות פרטיים, שמות משפחה, רחובות, ערים, מילים בגודל בינוני שנלקחו ממילון, ונכתבו גם בסדר הפוך, מהסוף להתחלה, מספרי רכב, ומחרוזות קצרות של ספרות אקראיות. אזי הם הצפינו כל אחד מהנ"ל בהשתמשם באלגוריתם של סיסמא מוצפנת. ובדקו אם אחת מהסיסמאות המוצפנות מתאימה לחלוטין לרשימה שלהם. מעל 86% מהסיסמאות נמצאו ברשימה שלהם.

אם כל הסיסמאות מורכבות מ- 7 תווים אקראיים מתוך 95 תווי אסקי, גודל החיפוש הוא 95 בחזקת 7. בקצב של 1000 הצפנות בשניה, זה ייקח 2000 שנה לבנות רשימה שתבדוק את הסיסמאות. יותר מזה, הרשימה תתפוס 20 מליון סרטים מגנטיים. אם נדרוש סיסמאות, שיכילו לפחות אות קטנה אחת, אות גודלה אחת, תו מיוחד אחד, ותהיה מורכבת מ- 7 תווים נשפר מאוד הסיסמאות. גם אם זה נחשב בלתי אפשרי לדרוש מהמשתמש לבחור סיסמא הגיונית, המחקר תיאר טכניקה שמקשה על הפיצוח והיא לחבר N ביטים של מספרים אקראיים, עם כל סיסמא, כאשר שניהם נבדקים בכניסה. המספר האקראי משתנה בכל פעם שהסיסמא משתנה. המספר האקראי מאוחסן בקובץ הסיסמאות, באופן בלתי מוצפן, כך שכולם יכולים לקרוא אותו, במקום סתם לאחסן את הסיסמא המוצפנת בקובץ הסיסמאות, הסיסמא, והמספר האקראי משורשרים ואז מוצפנים יחד. התוצאה המוצפנת מאוחסנת בקובץ הסיסמאות.

<u>עמוד 190</u>

חשוב על הסיבוכים שיהיו לפורץ שרוצה לבנות רשימה של סיסמאות דומות, להצפין אותן, לשמור את התוצאות בקבצים ממוינים, כך שכל סיסמא מוצפנת תוכל להימצא בקלות. אם הפורץ חושד ש"מרילין" עלולה להיות הסיסמא, לא מספיק לו להצפין "מרילין", ולאחסן את התוצאה, הוא צריך להצפין 2 בחזקת N מחרוזות כמו "מרלין 000" וכדומה.

הטכניקה הזאת מגדילה את האפשריות ב- 2 בחזקת N. UNIX משתמשים בשיטה זו עבור N=12. וזה ידוע בשם SALTING. "תיבול" קובץ הסיסמא. ורסיות אחרות של UNIX הופכות את קובץ הסיסמאות עצמו, לבלתי קריא, אבל מספקות תוכנית שתחפש כניסות לפי בקשה, ומאפשרות דחייה מספקת על מנת לעכב כל תוקף. למרות ששיטת ההגנה הזאת נגד פולשים שמנסים לחשב רשימות גדולות של סיסמאות מוצפנות עושה מעט מאוד להגן על משתמש בשם דוד שהסיסמא שלו דוד. כלומר המשתמש הפשוט, שהסיסמא שלו פשוטה. שיטה נוספת היא לעודד אנשים לבחור סיסמאות טובות, היא להציע סיסמאות על ידי המחשב. למחשבים מסוימים יש תוכניות המייצרות מילים חסרות משמעות, שקלות לביטוי, שיכולות להיות סיסמאות טובות.

מחשבים אחרים מבקשים מהמשתמש לשנות את הסיסמאות באופן קבוע, ובכך להקטין את הנזק אם הסיסמא דולפת החוצה. הדוגמא הקיצונית לשיטה זו היא סיסמא חד פעמית. כאשר משתמשים בסיסמא חד-פעמית, המשתמש מקבל ספר שמכיל רשימת סיסמאות. בכל כניסה הוא משתמש בסיסמא אחרת מהרשימה. אם הפורץ מגלה את הסיסמא, היא לא תעזור לו, מכיוון שבפעם הבאה הוא יצטרך סיסמא אחרת. מומלץ מאוד שהמשתמש המנע מלאבד את ספר הסיסמאות. זה כמעט ברור שכאשר הסיסמא מודפסת, המחשב לא יראה את התווים המודפסים. על מנת להסתיר אותם מעיניים זרות, בסביבות המסוף. מה שפחות ברור הוא שאסור לאחסן סיסמאות בצורה בלתי מוצפנת במחשב, ואפילו לא בהנהלת מרכז המחשבים, לא יהיו סיסמאות בלתי מוצפנות. שמירת סיסמאות בלתי מוצפנות, בכל מקום, יוצרות בעיות רבות.

וריאציה נוספת לרעיון הסיסמא הוא לתת רשימה ארוכה של שאלות, והתשובות יוצפנו לאחר מכן במחשב. השאלות יבחרו כך שלא יהיה צורך לרשום את התשובות באיזה שהו מקום. בכניסה למערכת המחשב ישאל אחת מהשאלות באופן אקראי, ויבחן את התשובה.

עמוד 191

שיטה אחרת CHALLENGE RESPONSE כאשר זה בשימוש המשתמש בוחר אלגוריתם לדוגמא X בחזקת 2. כאשר הוא נכנס למערכת המחשב מדפיס ארגומנט, לדוגמא, 7 ואז המשתמש מדפיס 49. האלגוריתם ניתן לשינוי על ידי המשתמש בכול זמן. פומביות האלגוריתמים אינה עומדת בסתירה לדרישות הבטיחות היות וכך נמנעת הסתמכות על סודות בעת תכנון מערכת הבטיחות. כפי שכבר הדגשנו קודם, מנגנוני הגנה פומביים בטוחים יותר.

זיהוי פיסי

הרעיון הוא עתיק יומין: אם המכונה תוכל לזהות בוודאות את המשתמש ממש כשם שאדם עושה זאת, לא יהיה צורך במנגנוני הגנה כגון סיסמא. יתירה מזו, גם הפושע יזוהה אוטומטית על ידי המכונה, דבר שמלכתחילה יקטין מאוד את מספר ניסיונות הפריצה למערכת. למרבה הצער, בשלב זה אנו עדיין רחוקים מאוד מהאפשרות לחקות את יכולת הזיהוי האנושית. לפיכך, המכונה מנסה לזהות את האדם על פי תכונות פיסיות הניתנות לזיהוי אוטומטי. לזיהוי הפיסי ישנן גם מגבלות חברתיות כבדות. זוהי כנראה גם הסיבה לכך שרוב המערכות מסתמכות על זיהוי באמצעות ידע (סיסמא), לעיתים בשילוב עם הוכחת בעלות על התקן נוסף (כרטיס מגנטי).

שיטה שונה לחלוטין היא בדיקה האם למשתמש יש חפץ, בדרך כלל כרטיס מגנטי מסוים. כרטיס זה מוכנס למסוף שבודק איזה כרטיס זה. שיטה זו יכולה להיות משולבת עם סיסמא, כך שהמשתמש יוכל להיכנס עם א. כרטיס ב. סיסמא. מכונות כגון בנקומט עובדים בשיטה זו.

שיטה נוספת היא מדידת אפיונים פיזיים שקשה לזייפן, כגן טביעת אצבעות, טביעת קול שהמחשב משווה מול נתונים שנמצאים אצלו.

שיטה אחרת היא בדיקת חתימה. המשתמש חותם את שמו עם עט מיוחדת שמחוברת למסוף, והמחשב משווה לחתימה שמאוחסנת אצלו.

שיטה טובה יותר היא לא להשוות את החתימה עצמה, אלא את תנועות העט, שנעשות בעת החתימה. מכיוון שזיפן טוב יוכל לזיף את החתימה, אבל יתקשה מאוד לחתום בדיוק באותן תנועות יד.

שיטה נוספת היא בדיקת אורך אצבעות. בשיטה זו לכל מסוף יש מתקן כפי שמצויר בשרטוט 26-4. המשתמש מכניס את היד לתוך המתקן, והאורך של כל אצבעותיו נמדד ומושווה מול מאגר הנתונים.

עמוד 192

אמצעי זהירות

מערכות מחשב הרציניות בקשר לביטחון, בדרך כלל מתעוררות יום אחרי שהם נפרצות, ונגרם להן נזק ומנסות שיטות הגנה כגון:

1. כל משתמש יוכל להיכנס למערכת רק ממסוף מסוים, ורק בזמנים מסוימים, ביום, שבוע חודש וכו'.
2. אבטחה מפני חדירה על ידי הטלפון ניתנת לביצוע בשיטה הבאה: כל אחד יכול לחייג ולהיכנס, אולם לאחר כניסה מוצלחת המערכת מייד מנתקת את השיחה, ומתקשרת אליו למספר הידוע לה מראש. משמעות שיטה זו היא שפורץ לא יוכל רק לפרוץ מכל קו טלפון, אלא רק מקו הטלפון של המשתמש.
3. בכל מקרה אם או בלי חיוג בחזרה, המערכת תיתן לפחות 10 שניות להדפיס כל סיסמא שתודפס על ידי המחייג, ותלך ותוריד את זמן הבדיקה, לאחר כל ניסיון כניסה בלתי מוצלח. לאחר שלושה ניסיונות בלתי מוצלחים, הקו ינותק ל- 10 דקות לפחות, ותישלח הודעה לקצין הביטחון. בקיצור מניעת כניסה לאחר ניסיון כושל של מספר סיסמאות.
4. כל הכניסות יוקלטו. כאשר משתמש נכנס למערכת המערכת תדווח את השעה והתאריך הכניסה האחרונה. זאת כדי לאתר כניסות לא מורשות.

5. "לארוב לפולש" לפרוש מלכודות. יש להחזיק כניסה אחת הקלה לאיתור ואז לעקוב אחרי המשתמשים בה. כלומר להחזיק LOGIN פשוט יחד עם סיסמא פשוטה, הקלה לאיתור.

עמוד 193

4.5 מנגנוני הגנה

שוב חוזרת ההבחנה החשובה בין מכניזם (מנגנונים) לבין מדיניות (דרך ההפעלה של המנגנונים).

4.5.1 הגנה על ידי קביעת תחומים – PROTECTION DOMAINS

מערכת המחשב מורכבת מהרבה אובייקטים שצריך להגן עליהם, כגון חומרה: לוח אם, זיכרון, מסופים, דיסק קשיח, מדפסות או תוכנות כגון מעבדים, קבצים בסיסי נתונים וכדומה. לכל אובייקט יש שם מיוחד אליו אנו מתייחסים, וסדרה של פקודות שמשמשות להפעלתו. READ AND WRITE מופעלות לגבי קבצים. UP AND DOWN לגבי סמפור.

מנגנון זה דורש שילוב של עקרונות יסוד כבר בשלב התכנון הראשוני של המערכת. לשם כך אנו קובעים תחום לכל אובייקט. תחום הוא סט של זוגות – אובייקט, וההרשאות שיש לתהליך ביחס לאובייקטים אחרים במערכת. העצמים יכולים להיות מכל סוג שהוא, חומרה או תוכנה.

אובייקט מסוים יכול להשתתף בכמה תחומים, כאשר בכל תחום יש לו הרשאות שונות. בכל רגע נתון, כל תהליך רץ בתחום מסוים של הגנה – כלומר יש אוסף של אובייקטים שהוא יכול לגשת אליהם בהרשאות מסוימות. תהליך יכול לעבור מתחום למשנהו בעת הריצה.

ב- UNIX התחום נקבע על פי מספרי ה- UID וה- GID של תהליך. עבור בדיוק אותה קומבינציה של מספרים נקבל בדיוק אותו תחום. בנוסף לכל תהליך ב- UNIX יש שני חלקים – חלק של המשתמש וחלק של המערכת – KERNEL. לחלק המערכת יש הרשאות נרחבות יותר ולכן כאשר מתבצעת קריאת מערכת תחילה היא מבצעת שינוי תחום, ורק אחר כך ה- KERNEL נכנס לפעולה.

עמוד 194

גם ביצוע קריאת המערכת EXEC המדליקה את סיביות ה- SETUID או SETGID, יוצרת מספרי UID ו- GID חדשים ולכן עוברת לתחום חדש.

ב-MULTICS התמיכה היא לא רק בשני חלקים – משתמש ומערכת ומעבר תחום ביניהם, אלא ב- 64 תחומים. תהליך יכול להיות מורכב מפרוצדורות שרצות ולכל אחת תחום משלה הנקראות RINGS. לתחום המערכת – KERNEL היו את ההרשאות החזקות ביותר (שרטוט 28-4). ככל שמתקרבים לגרעין, ההרשאות חזקות יותר.

כאשר פרוצדורה אחת קראה לפרוצדורה אחרת ב- RING אחר, מתבצעת מלכודת, על מנת לתת למערכת שהות להחליף תחום הגנה.

שאלה חשובה היא – איך המערכת מנהלת מעקב אחרי: איזה עצם שייך לאיזה תחום?

עמוד 195

פתרון אחד הוא להחזיק מטריצה שהשורות בה מהוות את התחום והעמודות עצמים. התוך בכל תא מציין את ההרשאות (שרטוט 29-4).

תחומים שמחליפים את עצמם, כמו ב-MULTICS יכולים להיכלל במטריצה. כלומר תחום הוא גם עצם (ועליו מוגדרת פעולת ENTER). איור 30-4. באיור נוספו 3 תחומים למטריצה, כמו אובייקטים. תהליך בתחום 1 יכול לעבור לתחום 2, אולם משם אין דרך חזרה. המצב דומה ב- UNIX באופן ביצוע תוכנית SETUID. שום החלפת תחומים אחרת לא מורשת בדוגמא.

4.5.2 רשימות גישה – ACL

למעשה האחסון של המטריצה בשרטוט 30-4 מתבצע לעיתים רחוקות, היות והמטריצה גדולה ודלילה. לרוב התחומים אין גישה, בכלל לרוב האובייקטים, כך שאחסון במטריצה גדולה וריקה זה בזבוז שטח דיסק.

ישנן שתי שיטות מעשיות לאחסון מטריצה, לפי שורות ועמודות, ולאחר מכן לאחסן רק את האלמנטים הלא ריקים. שתי השיטות שונות אחת מהשניה באופן מפתיע. בפרק זה נתבונן באחסון בעמודות, ובפרק הבא, באחסון בשורות.

<u>אחסון על פי עמודות</u>

עבור כל אובייקט תשמר רשימה מסודרת של התחומים הקשורים אליו (יכולים לגשת אליו). רשימה זאת נקראת Access Control List.

<u>עמוד 196</u>

אם ניישם את השיטה ב-UNIX הדרך הטובה ביותר תהיה לשים את רשימת ה-ACL של כל קובץ בבלוק נפרד בדיסק, שמספרו נרשם בטבלת ה- I-NODE של הקובץ. מכיוון שרק החלקים הלא ריקים של המטריצה, מאוחסנים, השטח שיידרש לכל רשימות ACL קטן בהרבה ממה שהיה צריך בשביל לאחסן את כל המטריצה. לשם הדגמה לאופן פעולת ACL ב-UNIX היכן שהתחום מוגדר על ידי זוג (UID, GID). למעשה משתמשים ב-ACL ב- UNIX ו-MULTICS פחות או יותר בדרך שתתואר להלן, כך שהדוגמא לא לגמרי היפוטית.

נניח שיש לנו 4 Maaile, Jelle, Els, Jan . משתמשים ששיכים לקבוצות מערכת, צוות סטודנט וסטודנט בהתאמה. נניח שבחלק מהקבצים יש את רשימת ACL הבאים:

File0: (jan,*,RWX)
File1:(jan,system,RWX)
File2:(Jan,*,RW-),(Els,staff,R--),(Maaike,*,RW-)
File 3:(*,student,R--)
File4:(Jelle,*,---),(*,student,R--)

כל כניסת ACL, שבסוגריים מצביעה על UID, GID והרשאת כניסה (Read, Write, Execute). כוכבית פרושה כל UID או GID. Flie0 ניתן לקריאה, כתיבה או ביצוע על ידי כל תהליך עם UID=Jan, וכל GID. File1 ניתן לגישה רק על ידי פרוצדורה עם UID=Jan ו- GID=SYSTEM. התהליך שבו UID=Jan ו- GID=STAFF יכול לאפשר גישה ל- File0, אבל לא ל- File1. אפשר לקרוא או לכתוב את File2 על ידי תהליך עם UID=Jan וכל GID אחר שנקרא על ידי תהליך עם UID=Els ו- GID=STAFF או על ידי תהליך UID=Maaike וכל GID. אפשר לקרוא את File3 על ידי כל סטודנט. File4 מעניין במיוחד. כי אין גישה בכלל לכל אחד עם UID=Jelle בכל קבוצה, אבל כל שאר הסטודנטים יכולים לקרוא אותו. על ידי שימוש ב-ACL ניתן לאסור על UIDS או GIDS מסויימים לגשת לאובייקט, ובאותו זמן לאפשר לכל אחד אחר באותה הקבוצה את הגישה.

עד כה דיברנו מה UNIX לא יכול לעשות, עתה נראה מהו כן יכול לעשות. על ידי 3 ביטים, RWX לכל קובץ בשביל הבעלים, קבוצת הבעלים, ואחרים. הסכמה הזאת דומה ל-ACL אבל מכווצת ל- 9 ביטים. זוהי רשימה מקושרת לאובייקט, המפרטת את הרשאות הגישה. בעוד שסכמת ה- UNIX של 9 ביטים היא בברור כללית משיטת ה-ACL, בפועל היא מדויקת והיישומים שלה יותר פשוטים וזולים. הבעלים של האובייקט יכול לשנות את ה-ACL בכל זמן וכך לאסור גישה על אלה שהיו מאופשרים קודם לכן, באופן קל יותר.

הבעיה היחידה היא שבשינוי ה-ACL, כנראה לא תשפיע על משתמשים שכרגע משתמשים באובייקט, לדוגמא, פתחו את הקובץ.

4.5.3 <u>רשימות יכולת – C-LISTS</u>

השיטה השניה היא לפי שורות. כאן אנו שומרים רשימות הפוכות מאלו שתיארנו בסעיף הקודם: לכל תהליך מוצמדת רשימת העצמים שאליהם הוא יכול לגשת וכמו כן כל ההרשאות למימוש גישה זו. כל רשימה כזו נקראת רשימת היכולות של התהליך. כך בכל פעם שתהליך מבקש לבצע פעולה על עצם, נסרקת רשימת היכולות שלו, והפעולה מתאפשרת אך ורק אם העצם אכן מופיע ברשימה זו והפעולה הנדרשת מותרת.

מדוע דרושה הגנה על רשימת היכולת של התהליך מפני בעל התהליך עצמו?
כל בעל עניין חשוד מראש בכך שירצה להרחיב את סמכויותיו על דעת עצמו. לכן התהליך הוא החשוד הראשי בניסיון להגדיל את רשימת היכולות שלו עצמו.
רשימת יכולות טיפוסית נראית באיור 4-31. בכל יכולת יש שדה TYPE אשר אומר איזה סוג של אובייקט זה. יש שדה שנקרא RIGHTS שהוא מפת ביטים המציינים מי מהמשתמשים החוקיים של סוג אובייקט זה מורשה, ושדה "אובייקט" שהוא מצביע על האובייקט עצמו, לדוגמא מספר I-node.

עמוד 197

רשימת יכולת הן בעצמן אובייקטים ואפשר להצביע עליהן מרשימות יכולות אחרות, ובכך להתחלק בתת תחומים משותפים. רשימת יכולת – C-LIST חייבות להיות מוגנות מהתערבות של המשתמש. 3 שיטות מוצעות להגן עליהם.

1. TAGGED ARCHITECTURE – מבנה תגיות – שהוא פתרון בחומרה שבכל מילה בזיכרון יש ביט מיוחד המעיד על המצאות רשימת יכולת, וניתן לשנות אותו רק על ידי תוכנית שרצה בגרעין, כמו מערכת ההפעלה ולא על ידי התהליך עצמו.
2. שמירת ה-C-LISTS בתוך מערכת ההפעלה ופניה של תהליך על ידי מספר מזהה.
3. שמירת ה- C-LIST במרחב משתמש אבל להצפין כל יכולת עם קוד סודי לכל רשימה שלא ידוע למשתמש.

בנוסף לאפשריות האובייקט כגון READ AND EXECUT יש 4 יכולות נוספות:

1. העתק יכולת: צור יכולת חדשה לאותו אובייקט.
2. העתק אובייקט: צור עותק של אובייקט עם יכולת חדשה.
3. הסר יכולת: מחק כניסה מה-C-LISTS, האובייקט לא מושפע.
4. השמד אובייקט: הסר לקביעות אובייקט ויכולת.

שיטות רבות של מערכות יכולת מאורגנות כאוסף של מודולים. עם TYPE MENEGER MODULS לכל אובייקט בנפרד. בקשת לפעולה מסוימת על קובץ מסוים נשלחת למנהל הקבצים, בעוד שבקשות לעשות פעולות שקשורות לתיבת הדואר מופנות למנהל תיבת הדואר.

עמוד 198

שיטת ה-C-LIST מעודדת סיווג של העצמים במערכת. כל העצמים שניתן לבצע עליהם פעולות זהות (ללא התחשבות בהרשאות הספציפיות) מסווגים יחדיו. לכל סוג כזה ישנו תהליך המסוגל לבצע את כל הפעולות האפשריות על סוג העצמים שבהם הוא מטפל. דוגמא לכך הוא מנהל מערכת הקבצים, היכול לבצע את כל הפעולות האפשריות על קבצים (מחיקה, יצירה קריאה וכו').
בקשות אלו מלוות עם היכולת הרלוונטית. הבעיה שמתעוררת כאן, מכיוון שסוג מנהל המודול הוא תוכנית רגילה. המשתמש בקובץ יכול לבצע חלק מהפקודות על הקובץ, אבל לא יכול לקבל תצוגה פנימית שלו כגון ה- I-NODE שלו. זה חיוני שמנהל של קבוצת עצמים צריך להיות בעל הרשאות הכוללות את כל ההרשאות הספציפיות שיש לתהליך העשוי לפנות אליו. אלא שזהו כוח רב מאוד ויש למנוע אפשרות שהתהליך הפונה ינצל זאת לרעה. לכאורה, העתקה פשוטה של יכולות התהליך הפונה היתה פותרת את הבעיה, אולם לצורך מימוש הבקשה (ללא חריגה מהיכולות עצמן) יש צורך בכוח רב יותר: כדי לפתוח קובץ שלא באמצעות קריאת מערכת, יש צורך בגישה למבני נתונים של מערכת הקבצים שלתהליך רגיל אין ואסור שתהיה גישה אליהם. לפיכך יש צורך בתכנון זהיר מאוד כאשר כותבים תכנית ליישום מנהלי קבוצות עצמים.
בעיה נוספת היא – איך לבטל גישה לאובייקט מסוים?
קשה למערכת למצוא את כל היכולות החיצוניות לכל אובייקט, מכיוון שהם אוחסנו ב--C LISTS בכל רחבי הדיסק.

1. כל רשימת יכולת תצביע לאובייקט באופן עקיף דרך אובייקט אחר. ואז המערכת תשבור את הקשרים האלה – בין אובייקט המקשר ליכולת האובייקט (האובייקט המקשר יצביע לאובייקטים האמיתיים).

2. במערכת AMOBA כל אובייקט מכיל מספר אקראי שמיוצג גם ברשימת היכולת. בכל פנייה מספרים אלה מושווים. בעל האובייקט יכול לבקש לשנות את המספר האקראי ובכך לבטל את הגישה אליו.

4.5.4 מודלים להגנה

טבלת הרשאות (טבלאות המכילות מידע על תחומים) כפי שמתוארות בציור 4-29 אינן סטטיות. הן משתנות לעיתים תכופות. כאשר נוצרים אובייקטים חדשים, אובייקטים ישנים מושמדים, והבעלים מחליט להוריד או להגביל סט של משתמשים מהאובייקטים שלהם. נתנה תשומת לב רבה למודלים של הגנה שבהם טבלת ההרשאות משתנה בתכיפות. אובחנו 6 פעולות פשוטות על טבלת ההרשאות שיוכלו לשמש בסיס לכל מערכת הגנה.

1. צור אובייקט
2. בטל אובייקט
3. צור תחום
4. בטל תחום
5. עדכן הרשאות
6. הסר הרשאה

כאשר מסתכלים על טבלת ההרשאות עצמה כעל עצם, ניתן להגדיר עליה 6 פעולות המספיקות כדי להתאימה לדרישות שונות של התהליכים. הטבלה ו-6 הפעולות המוגדרות עליה מספקות את המנגנון להגנה על המערכת. דרך השימוש בהן היא המדיניות. עד לפני שקובעים מדיניות, עולה השאלה העקרונית מהם התנאים שמכתיבה מדיניות בטוחה והאם מדיניות כזאת אפשרית בכלל, בהינתן קבוצת פעולות כגון זו שמוגדרת כאן.

מדיניות בטוחה – מדיניות שבה כל פעולה מותרת מעבירה את התהליך למצב מותר בטבלת ההרשאות.

פקודות אלו נקראות פקודות הגנה, והן מאפשרות לתוכניות משתמש לשנות את הטבלה. הן לא יבצעו את הפקודות באופן ישיר. לדוגמא, אם למערכת יש פקודה ליצירת קובץ, הבודקת קודם אם הקובץ קיים, ואם לא יוצרת אובייקט חדש, ונותנת לבעלים את ההרשאות לקובץ. כמו כן יכולה להיות פקודה המרשה לבעלים להעניק הרשאה לכל אחד במערכת לקרוא את הקובץ, היא

למעשה הכנסת הרשאת קריאה בכניסות הקובץ החדש בכל תחום.

עמוד 199

בכל רגע, המטריצה קובעת מה יכול לעשות כל תהליך השייך לאיזה תחום, ולא מה הוא מוסמך לעשות. המטריצה היא מה שנכפה על המערכת. הרשאות קשורות למדיניות.

כדוגמא להבחנה, באיור 4.32 שבה תחום קשור למשתמש. בחלק A באיור אנו רואים מדיניות הגנה (דומה ל- read and write ב- UNIX). הנרי יכול לקרוא ולכתוב ב- mailbox7, רוברט יכול לקרוא ולכתוב secret, וכל שלושת המשתמשים יכולים לקרוא ולבצע compiler. כעת נדמיין שרוברט מאוד חכם, ומצא דרך לאשר פקודות המשנות את המטריצה, והיא שונתה לחלק B באיור. כעת הוא השיג גישה ל- mailbox7, דבר שהוא לא מוסמך לו. אם הוא ינסה לקרוא את הקובץ מערכת ההפעלה תוציא לפועל את בקשתו כי אינה יודעת שההרשאה שלו בלתי חוקית. צריך כעת להיות ברור שניתן לחלק את הטבלה לשתי טבלאות, אחת למצבים מוסמכים, ואחת ללא מוסמכים.

השאלה הנשאלת היא האם מכניזם ההרשאות (פקודות ההגנה) אכן אוכף את מדיניות ההגנה. כדוגמא פשוטה נניח מערכת אבטחה שנמצאת בשימוש הצבא. כל אובייקט הוא: לא מסווג, שמור, סודי וסודי ביותר. כל תחום (ולפיכך כל תהליך) גם הוא שייך לארבעת דרגות הבטיחות. למדיניות ההגנה יש 2 חוקים:

1. תהליך לא יכול לקרוא אובייקט בדרגה גבוהה משלו, אולם הוא יכול לקרוא כל אובייקט בדרגה נמוכה משלו.
2. תהליך לא יכול לכתוב באובייקט שדרגתו נמוכה משלו.

עמוד 200

על ידי נתינת, מדיניות, מצב תחילי של המטריצה (כולל דרך כלשהי לומר איזה אובייקט היא באיזה דרגה), וסדרה של פקודות לשינוי המטריצה, אזי נרצה דרך להוכיח שהמערכת בטוחה. הוכחה זו קשה להשגה. יישומים כללים של המערכת אינם בטוחים באופן תאורתי.

4.5.5 ערוצים חסויים

למרות כל מנגנוני הבטיחות, נראה שבמערכת מרובת תהליכים, אין שום דרך המאפשרת למנוע משני תהליכים להעביר ביניהם מידע שאותו הם אינם רשאים להעביר. במילים אחרות, כל שני קושרים או יותר יכולים לנהל ביניהם דיונים ולזמום תכניות משותפות על אפה ועל חמתה של מערכת הבטיחות, ואין דרך למנוע זאת מהם.

אם כן מה ערכה של מערכת הבטיחות?

מנגנוני הבטיחות שתיארנו עד כה מבטיחים דבר אחד בצורה ודאית למדי: תהליך אינו יכול להשתלט על משאבים שהוא אינו מורשה לקבלם. לעומת זאת, אין שום דרך למנוע העברה של מידע בין שני תהליכים המשתפים ביניהם פעולה.

בחלק הקודם ראינו כיצד ניתן ליצור מודלים להגנת המערכת. בחלק זה נראה כמה חסר תועלת ליצור מודלים אלו. ובפרט נראה שאפילו שלמרות שמערכת הוכחה כבטוחה באופן אבסולוטי, יש דליפת מידע בין תהליכים שבאופן תיאורטי אסורים בתקשורת. הרעיון העומד מאחורי הדברים הנ"ל על שם LAMPSON הוא שישנם 3 תהליכים. התהליך הראשון הוא הלקוח (CLIENT) שרוצה שעבודה מסוימת תבוצע על ידי תהליך שני, והוא השרת (SERVER). הלקוח והשרת אינם "בוטחים" זה בזה. לדוגמא, עבודתו של השרת היא לעזור ללקוח למלא את טפסי מס הכנסה. הלקוח חושש שמה השרת יעתיק את הנתונים הכספיים. לדוגמא ישמר רשימות סודיות של משכורות, ואז ימכור את הרשימה. השרת מאידך חושש שמה הלקוח יגנוב את תוכנת השכר שלו.

התהליך השלישי הוא משתף פעולה עם השרת – הוא עוזר לשרת "לעבוד" על הלקוח. שני תהליכים אלה שרת ומשתף פעולה הם שייכים לאותו בעלים (שרטוט 33-4).

מטרת התרגיל היא לתכנן מערכת שבה זה בלתי אפשרי שהשרת ידליף למשתף הפעולה מידע שהועבר בצורה לגיטימית מהלקוח. בעיה זו מכונה בעיית הריתוק – CONFINEMENT PROBLEM. כלומר המטרה היא למנוע מהשרת להעביר מידע למשתף פעולה, כלומר לגשת אליו וזאת על ידי מטריצת הגנה (כפי שתוארה) שבה משתף הפעולה יש הרשאה רק להיקרא. אך מסתבר ששני תהליכים אלו יכולים להעביר אינפורמציה בינים על ידי ערוץ חסוי COVERT CHANNEL. השרת יכול לתקשר עם מחרוזת בינרית שגם המשתף פעולה יקרא אותה. אופן התקשורת הוא קידוד בינרי בין השרת למשתף הפעולה. ערוץ זה הוא "רעשני" במידה מסוימת מכיוון שהוא מכיל מידע נוסף שלא לצורך אך ניתן לשלוח אינפורמציה על ערוץ "רועש" תוך שימוש קוד מתקן טעויות (למשל קוד המינג).

לא ניתן למנוע את התקשורת דרך ערוץ חסוי על ידי המודל הבנוי על מטריצת תחומים. כל דרך איתות היא ערוץ חסוי, למשל בקשת משאבים ושחרורם, יכול לשמש כאיתות.

השרת דורש משאב כדי לשלוח 1, ומשחרר אותו כדי לשלוח 0. ב-UNIX השרת יכול ליצור קובץ כדי לאותת 1, ולשחרר אותו, כדי לאותת 0. משתף הפעולה יכול להשתמש בקריאת מערכת ACCESS כדי לבדוק האם הקובץ קיים. לדאבוננו, ישנם ערוצים חסויים רבים אחרים שקשה מאוד לאתרם, ולמעשה קשה מעשית לעשות משהו בנדון. דוגמא נוספת היא, למשל מותר לשרת לדווח לבעלים שלו מהו החשבון שיש לשלוח ללקוח. אם החשבון הוא 100$ וסך המשכורת היא 53K$, ישלח החשבון 100.53$ לבעלים.

דוגמאות לשימוש בערוץ חסוי – וויסות CPU, דפדוף. בדפדוף 1 יראה פסיקת דף, ו- 0 להיפך.

עמוד 205

פרק 5 : קלט\פלט

ללא קלט ופלט אין כל ערך למחשב. במצב זה, רק טבעי הוא כי חלק גדול מאוד ממערכת ההפעלה מיועד לטיפול בהתקני קלט ופלט ובתיאום הפעילות ביניהם. אחת המטרות החשובות בהקשר זה היא כפי שכבר הזכרנו, הסתרה מוחלטת של מגוון סוגי החומרה הקיימים להתקני קלט\פלט תחת מעטה של ממשק אחיד. הסיבות להסתרת חומרה:

א. הסתרת החומרה מקלה על כתיבת תכניות קלות להעברה.

ב. ממשק אחיד מקטין חלק גדול מהמעמסה המוטלת על המתכנת.

ג. הסתרת החומרה מונעת אפשרות של תהליך להשתלט עליה לצרכיו.

יש למצות ככל האפשר את התקני קלט\פלט. תעסוקתם המלאה של התקני קלט\פלט מבטיחה גם מקביליות טובה יותר. כיצד?

להתקני קלט\פלט צמודים בקרים שיכולים לפעול במידה מסוימת של עצמאות בלי להזדקק ישירות למעבד הראשי. כך למשל, יכול בקר הדיסק להעביר במשך זמן מסוים נתונים ישירות לזיכרון המחשב. במשך זמן זה, המעבד פנוי לשרת תהליכים נוספים.

5.1 חומרה של התקני קלט פלט

עם כל רצוננו להסתיר את החומרה, הסתרה זו אפשרית רק הודות להכרת התכונות המיוחדות של ההתקנים. התכונות הפיסיות האלה מנוצלות ככל האפשר ליצירת הממשק האחיד. בהמשך נסקור חלק מתכונות אלו.

אנו מתרכזים באיך החומרה מתוכנתת ולא איך היא פועלת מבפנים. למרות זאת תכנות התקני קלט\פלט קשור לתכונות הפיסיות.

עמוד 206

5.1.1 התקני קלט\פלט

באופן כלל ניתן לחלק את התקני הקלט\פלט לשתי קטגוריות:

1. BOLCK DEVICES – התקנים המאחסנים מידע בבלוקים קבועים, לכל אחד הכתובת שלו. בדרך כלל גודל הבלוקים נע בין 128 ל- 1024 ביטים. ניתן לקרוא או לכתוב לכל בלוק באופן ישיר ועצמאי בלי קשר לשאר המידע\בלוקים. דיסקים הם בעצם התקני בלוק. (בהתאם לכך יש קובץ מתאים – BLOCK SPECIAL FILE). אם תסתכל מקרוב הגבולות בין התקנים, בעלי כתובות של בלוק, ואלה שהם לא, לא מוגדרים בצורה ברורה. כולם מסכימים שהדיסק הוא בלוק בעל כתובת, כיוון שלא משנה מיקום הזרוע ברגע זה, תמיד אפשר לבקש צילינדר אחר, ואז לחכות שהבלוק המבוקש יגיע אל מתחת לראש הקריאה. טייפים אינם מוגדרים כ- BOLCK DEVICE. למרות שאם מבקשים את הבלוק ה- N, אז לכאורה הטייפ יכול ללכת לאחור, ואז להתקדם עד שהוא מגיע לבלוק ה- N. זה כמו פעולת החיפוש של הדיסק אחר צילינדר, רק שהיא לוקחת יותר זמן.
2. CHARACTER DEVICE – התקן זה מקבל או מוסר רצף של תווים בלי להתייחס למבנה של הבלוק. אין לו כתובת, ולא מבצעים אתו פעולה חיפוש או רישום בכתובת מסוימת. לדוגמא, מסופים, מדפסות, סרטי ניר, כרטיסי ניר, עכבר, והתקנים נוספים אחרים שאינם מתנהגים כמו דיסק (בהתאם לכך קובץ – CHARACTER SPECIAL FILE).

החלוקה הנ"ל איננה מושלמת. חלק מההתקנים לא יכולים להיות מסווגים לאחת הקטגוריות, לדוגמא, שעון – אין לו כתובת בבלוק, יחד עם זאת הם לא מיצרים או מקבלים רצף של תווים. כל מה שהוא עושה לגרום לפסיקה בפרקי זמן מסוימים. (מיפוי זיכרון כנ"ל). למרות זאת שתי קטגוריות אלו מספיקות בדרך כלל לשימוש בסיסי של מערכת ההפעלה. לדוגמא, שיטת הקבצים.

מערכת הקבצים עוסקת רק ב- BLOCK DEVICE ומשאירה את החלק שלא מתאים לחומרה בדרגה נמוכה יותר שנקראת DEVICE DRIVERS.

DEVICE DRIVERS - תוכנית שיודעת לשלוט בהתקן מסוים כדי שהמשתמש לא יצטרך לדעת בדיוק איך פועל ההתקן ברמה הפיזית.

5.1.2 בקרים – DEVICE CONTROLLERS

היחידות הטיפוסיות לקלט\פלט מורכבות מרכיבים מכנים ורכיבים אלקטרונים. ניתן בדרך כלל להפריד את שני החלקים ובכך לתכנן באופן כללי יותר, ומודולרי. הרכיבים האלקטרוני נקראים DEVICE CONTROLLER – בקר, או מתאם. מערכת ההפעלה אינה פונה להתקן עצמו, שהוא התקן פיסי, אלא לכרטיס אלקטרוני מיוחד הנקרא בקר. הבקר מציג בפני מערכת ההפעלה ממשק פרימיטיבי לניהול כל פעולות הקלט\פלט האפשריות בהתקן זה. ממשק זה הוא פרימיטיבי כי הוא כולל רק כתיבה וקריאה של סדרות של סיביות לתוך אוגרים של הבקר, אך זוהי שפה המוכרת היטב למערכת ההפעלה. למעשה, אוסף הפקודות של הבקר יוצר שפה שבה יש לפנות אליו.
היתרון של כפיפות זו הוא עצום, הן עבור יצרני התוכנה היכולים לכתוב תכניות סטנדרטיות, והן עבור יצרני החומרה היכולים לייצר התקנים תואמים. בדרך כלל, מערכת ההפעלה אינה פועלת ישירות באמצעות שפה זו. תחת זאת יש במערכת ההפעלה תכנית תיאום מיוחדת עבור כל התקן (DEVICE DRIVER), שתפקידה לתרגם פקודות כלליות לשפה המובנת להתקן זה. כאשר מערכת ההפעלה מעוניינת לקרוא בלוק בדיסק, היא פונה אל המתאם (תכנית התיאום) שמתרגם עבורה את ההוראה לשפת הבקר.

עמוד 207

על גבי כרטיס הבקר יש בדרך כלל מחבר אליו מתחבר כבל המוביל להתקן עצמו. בקרים רבים יכולים לטפל ב- 2,4,8, התקנים זהים. אם הממשק בין הבקר להתקנים הוא ממשק סטנדרטי כגון ANSI, IEEE, ISO . החברות המייצרות יספקו בקרים הוא התקנים שיתאימו לממשק הזה. חברות רבות מייצרות, DISK DRIVER שמתאים לממשק בקרי דיסקים של IBM. ציינו את ההבחנה בין הבקרים להתקנים מכיוון שמערכת ההפעלה בדרך כלל תפעל עם הבקר, ולא עם ההתקן. כמעט כל המיקרו מחשבים משתמשים בערוץ בודד כפי שמצויר ב- 5-1, (ערוץ התקשורת בין המעבד, הזיכרון, הבקר והתקני הקלט\פלט) לתקשורת בין המעבד לבקרים. מחשבים גדולים משתמשים במודל אחר, רב ערוצים, שתוכננו במיוחד עבור מחשב קלט\פלט שנקראים ערוצי קלט\פלט שמורידים את העומס מהמעבד הראשי.
הממשק שבין ההתקנים לבקרים הוא בדרך כלל ממשק בדרגה נמוכה. לדוגמא, דיסק, אפשר לפרמט אותו ב- 8 סקטורים שבכל אחד 512 ביטים לערוץ. מה שיוצא מה- DRIVE הוא סדרה של רצף ביטים המתחילים במבוא, לאחר מכן סקטור של 4096 ביטים, ובסוף בדיקת סכום או קוד שגיאה (ECC). המבוא נכתב כאשר הדיסק מפורמט, ומכיל את מספר הסקטור, את גודלו, ומידע דומה לו. תפקידו של הבקר הוא להמיר סדרה של ביטים לבלוק של ביטים, ולטפל בכל התיקונים שצריך. בלוק הביטים בדרך כלל נבנה ביט אחרי ביט בתוך החוצץ שבתוך הבקר. אחרי ביצוע בדיקת הסכום והבלוק מוכרז שאין בו שגיאות, ניתן יהיה להעתיק אותו אל הזיכרון הראשי.
הבקר של מסוף CRT פועל גם הוא כהתקן של סדרות ביטים באותה רמה נמוכה. הוא קורא ביטים המכילים תווים שיוצגו מהזיכרון ומייצר סיגנלים שמווסתים את קרן ה- CRT הכותבת על המסך. הבקרים מיצרים סיגנלים המורה לקרן ה- CRT לחזור לתחילת שורה, לאחר שגמרה לסרוק את השורה, ולחזור לתחילת המסך לאחר שכל המסך נסרק.
ללא בקר ה- CRT היה צורך ליצר עבור מערכת ההפעלה תוכנית מיוחדת שתסרוק את שפורפרת המסך. בעזרת הבקר המערכת מאתחלת את הבקר עם מספר פרמטרים כגון, כמה תווים יש בשורה, וכמה שורות יש בכל המסך ונותנת לבקר עצמו לשלוט בקרן המסך.

עמוד 208

לכל בקר יש מספר מונים המשמשים לתקשורת עם המעבד. בחלק מהמחשבים מונים אלה הם חלק מכתובת הזיכרון הרגיל, התרשים נקרא MEMORY MAPPED I\O . מחשבים אחרים משתמשים בכתובת של מקום פנוי בזיכרון עבור קלט\פלט, כאשר לכל בקר מוקצה חלק ממקום זה. שרטוט 2-5 מראה את כתובת הקלט\פלט והקצאה של חלק מהבקרים במחשב IBM PC כדוגמא.

מערכת ההפעלה מתפעלת קלט\פלט על ידי כתיבת פקודות בתוך הבקרים. בקר הדיסק יכול לקבל לתוך האוגרים שלו 15פקודות שונות הקשורות לפעולות בדיסק. להרבה מהפעולות יש פרמטרים שמוטענים לתוך הבקר. כאשר הפקודה התקבלה (כגון קריאה, כתיבה, פרמוט חיפוש וכו') היא נכנסת לתוך הרגיסטר בבקר והמעבד יכול להמשיך בעבודתו. כאשר הפקודה מסתיימת הבקר גורם לפסיקה על מנת לאפשר למערכת ההפעלה לקבל שליטה על המעבד ולבדוק את תוצאת הפקודה. המעבד מקבל את התוצאה ואת סטטוס ההתקן על ידי קריאת בתים של אינפורמציה מהרגיסטרים בבקר.
במערכת ההפעלה UNIX תוכניות התאום הן חלק מגרעין המערכת ואילו ב- MS-DOS התוכניות מגיעות בנפרד.

יתרונות ההכללה בגרעין –

א. בגרעין כל הפעולות מהירות יותר.
ב. בגרעין ניתן להסתיר את תכונות החומרה.

חסרונות ההכללה בגרעין –

א. בכל החלפת בקר יש לחבר מחדש את הגרעין ביחד עם מתאם ההולם בקר זה.
ב. הגדלת הגרעין.

יתרונות הצבת המתאם מחוץ לגרעין –

א. ניתן להחליף את המתאם בקלות כאשר יש צורך בכך.
ב. קל יותר לבזר את המערכת.

חסרונות הצבת המתאם מחוץ לגרעין –

א. תנועת נתונים איטית יותר.
ב. קשה הרבה יותר למנוע שתהליך משתמש יפנה ישירות את הבקר.

5.1.3 <u>גישה ישירה לזיכרון – DMA</u>

בקרים רבים, במיוחד אלה המטפלים בהתקני בלוקים תומכים בגישה ישירה לזיכרון – DMA. על מנת להסביר כיצד עובד ה-DMA . תחילה הבקר קורא את הבלוק (סקטור אחד או יותר) באופן סדרתי ביט אחרי ביט עד שכל הבלוק נמצא בחוצץ הפנימי של הבקר. בשלב הבא הוא מחשב את הסכום לוודא שאין שגיאות בקריאה. לאחר מכן הבקר יוצר פסיקה. כאשר מערכת ההפעלה מתחילה לרוץ היא יכולה לקרוא את הבלוק מהחוצץ של הבקר, מילה בכל פעם, על ידי יצירת לולאה, כאשר בכל מחזור קריאה כזה המילה שנקראה מאוחסנים בזיכרון. באופן טבעי לולאה כזו שקוראת מילה בכל פעם מתוך הבקר מבזבזת זמן מעבד, ולכן הומצא ה-DMA על מנת לשחרר את המעבד מרמה נמוכה זו של עבודה. כאשר הוא בפעולה המעבד נותן לבקר שני סוגי אינפורמציה

א. הכתובת בדיסק של הבלוק שיש לקרוא ממנו.
ב. הכתובת בזיכרון אליה הבלוק צריך ללכת.
ג. מספר הבתים שיש להעביר (שרטוט 3-5).

עמוד 209

לאחר שהבקר קרא את כל הבלוק מההתקן למאגר הפנימי שלו, ווידא את סכומו לבדיקה שאין טעויות, הוא מעתיק את הבית הראשון אל הזיכרון הראשי לכתובת שמצוינת על ידי כתובת שסופקה על ידי ה- DMA. לאחר מכן הכתובת ב- DMA מועלית וסכך הכל מספר הבתים להעברה ב- COUNT ב-DMA מופחת ˙ (בהתאם למה שכבר הועבר). תהליך זה חוזר על עצמו עד שמונה הבתים באוגרי ה-DMA שווה ל- 0 ובזמן זה הבקר גורם לפסיקה ואז כשמערכת ההפעלה נכנסת לביצוע היא לא צריכה להעתיק את הבלוק לזיכרון – הוא כבר נמצא שם.
מדוע הבקר זקוק למאגר פנימי לאחסון הבלוקים שהוא מביא מהדיסק ואינו מעביר אותם ישירות לזיכרון? הסיבה היא שברגע שמתחיל מעבר מידע מההתקן, הבתים מגיעים בקצב קבוע ללא תלות בבקר, כלומר אם הוא מוכן או לא. אם הבקר ינסה לכתוב נתונים ישירות לזיכרון הוא יצטרך לעבור על כל ערוצי המערכת (שרטוט 3-5). עבור כל מילה שהוא רוצה להעביר. אם הערוץ עסוק, הבקר יצטרך להמתין ואז כאשר תשלח אליו מילה היא תצטרך

להישמר במקום כלשהו (אפילו כמה מילים אם הערוץ מאוד עסוק). כאשר הנתונים מועברים למאגר פנימי אין צורך בערוץ עד אשר ה- DMA מתחיל את פעולתו.
לתיאור הנ"ל של העברת הנתונים, העברה בשני שלבים דרך החוצץ, יש השפעה חשובה על ביצועי קלט\פלט:
בזמן שנתונים מועברים מהבקר לזיכרון, יש לדלג על בלוקים בדיסק על מנת לתת זמן לבקר להעביר את הנתונים לזיכרון – INTERLEAVING. גם שהדיסק מפורמט לוקחים זאת בחשבון. משמעות של בלוקים רצופים על הדיסק איננה בהכרח רציפות פיסית.
הרציפות מתבטאת הסדרת המקומות העוקבים שעליהם יתייצב הראש הקורא של הדיסק כשהוא מוכן לקריאה או לכתיבה.
ההשלכות של הבחנה זו על תכנית המנסה להגדיל את מהירות הקריאה באמצעות כתיבה מחדש של קבצים של סדרות רצופות של בלוקים היא שתכנית ציפוף איננה צריכה להתחשב בבעיית הרציפות הפיסית של בלוקים. בעיה זו נפתרת על ידי מספור הגזרות בעת ההפעלה החד-פעמית של תכנית אתחול (FORMAT) הדיסק. לפיכך, תכנית ציפוף צריכה רק להעתיק קבצים אל בלוקים שמספריהם רצופים. זוהי דוגמא לאלגנטיות שמתאפשרת על ידי הסתרת החומרה.

5.2 עקרונות למימוש קלט\פלט בתוכנה

למרות ההבדלים שבין ההתקנים הממשק המוצג למשתמש הוא זהה. ב-UNIX למשל, ניתן לגשת לכל ההתקנים על ידי פעולה אחידה: פתיחת הקובץ המיוחד המתאים להתקן. עקרונית, כל משתמש יכול לעשות זאת. במציאות, קבצים אלו מוגנים בדרך כלל ורק מנהל המערכת מורשה לגשת אליהם, וזאת מכיוון שקבצים אלו מאפשרים גישה ישירה לחומרה.

עמוד 211

5.2.1 מטרות תוכנת הקלט\פלט

הרעיון העומד מאחורי תכנון תוכנת הקלט\פלט הוא עצמאות ההתקנים – DEVIC INDEPENDENCE. יש לאפשר לכתוב תוכנית שקולטת נתונים מדיסק, דיסקט, מסוף וכנ"ל מוציאה נתונים מבלי לטפל בכך בתוכנית, או אפילו להעביר אותה קומפילציה שוב. על מערכת ההפעלה לטפל בכך ובנדרש כתוצאה מההבדלים בין ההתקנים.
UNIFORM NAMING – השמות הניתנים להתקנים צריכים להיות מחרוזת או מספר ולא צריכים להיות תלויים בהתקנים עצמם. יש לתת שמות אחידים. ב-UNIX ניתן לפנות להתקן מסוים על ידי:

א. פניה להתקן דרך הקובץ המייצג אותו.
ב. על ידי שימוש בהרכבת עצי קבצים הממומשת בפקודה MOUNT. כך ניתן להרכיב למערכת אחת עצי קבצים הנמצאים על דיסקים שונים. בדרך זו ניתן לגשת לקבצים הנמצאים על דיסק נוסף.

עקרון האי תלות של השמות הוא חשוב, לדוגמא ב- MS-DOS. במערכת זו שם ההתקן (למשל :A) מהווה חלק משמו שמלא של קובץ. הבעיה היא ששם ההתקן מצביע גם על מקומו הפיסי. :A ו- :B הם שמות של כונני תקליטונים. תכניות המשתמשות בשם המלא לא יפעלו על התקן אחר.
נושא חשוב נוסף בתוכנת קלט\פלט הוא טיפול בשגיאות. באופן כללי הטיפול בשגיאות צריך להיעשות קרוב ככל האפשר לרמת החומרה, כלומר ללא ידיעת המשתמש. ההתקן בו התרחשה הטעות ינסה לטפל בבעיה בעצמו (בבקר שלו). אם זה לא אפשרי ה-DEVICE DRIVER שלו ינסה ואם לא יצליח רק אז הטיפול בשגיאות יעלה לרמה גבוהה יותר – המשתמש והתוכנית שלו.
יש לשים לב כי יש סתירה בין הדרך הפיסית שבה מופעלות פעולות קלט\פלט לבין אופן הופעתן בתהליך. מבחינת התהליך, הפעולות הן תמיד סינכרוניות, גם כאשר מבחינה פיסית המצב שונה לחלוטין. מערכת ההפעלה מעוניינת כי פעולות אלו יהיו א-סינכרוניות וכך יתפנה המעבד לעבודה במקביל. לעמת זאת תהליך רגיל מעונין על פי רוב בפעולות קלט\פלט סינכרוניות, כלומר התהליך מעונין להמתין לביצוע פעולות קלט\פלט ורק לאחר מכן להמשיך.

רוב התכניות מתבססות על ההנחה שפעולות הקלט והפלט מתבצעות באופן סינכרוני היות וברוב המקרים המשך פעולת התכנית תלוי בקלט ולעיתים גם בפלט. במצב של פעולות א-סינכרוניות, הרצה חוזרת של אותה תכנית ועם אותם נתונים, עלולה לתת תוצאות שונות. הפתרון לדרישות המנוגדות נמצא כרגיל בתכונות המכונה המדומה. לגבי מערכת ההפעלה, פעולות קלט\פלט מתבצעות באופן א-סינכרוני, וכך מתאפשרת המקביליות האמיתית. לעומת זאת תהליכים רגילים נחסמים (הולכים לישון) עד לתום פעולות הקלט\פלט ומתעוררים עם השלמתן. כך, מבחינת התהליכים, הפעולות בוצעו באופן סינכרוני.

יש התקנים מסוימים כגון דיסקים הניתנים לשימוש במקביל על ידי מספר משתמשים בלי ליצור בעיות. אולם ישנם משאבים בלעדיים (DEDICTAED DEVICES), אשר בניגוד למשאבים הניתנים לשיתוף, אינם ניתנים לחלוקה בו-זמנית בין תהליכים שונים. הדוגמה המובהקת ביותר לכך היא המדפסת. התקן זה ניתן לשימוש רק על ידי משתמש אחד, ומערכת ההפעלה צריכה לדוגמא לפתור את בעיית ההדפסה כאשר מספר משתמשים שולחים הדפסות במקביל. דבר זה מסבך את מנגנוני החלוקה של מערכת ההפעלה.

עמוד 212

בסעיפים הבאים נעסוק בארבע שכבות מרכזיות המרכיבות את תוכנת ניהול ההתקנים המבצעים קלט\פלט:

א. מנהל הפסיקות
ב. מתאמי ההתקנים
ג. תוכנת קלט\פלט המשוחררת מאילוצי חומרה
ד. תוכנות קלט\פלט ברמת המשתמש

5.2.2 מנהל הפסיקות

חיוני להסתיר את פסיקות ההתקנים מעיני תהליך המשתמש משתי סיבות עיקריות:

א. הפסיקות הן תלויות חומרה באופן מובהק.
ב. אסור שלמשתמש תהיה גישה כלשהי לפסיקות החומרה של המערכת. גישה כזו עשויה למוטט בקלות את מערכת ההפעלה כולה.

בכל זאת תהליכים מסתמכים על פסיקות ואף עשויים לבקש במפורש פעולות הדורשות פסיקות, כגון הרדמה או הערה. אולם פעולות אלו לובשות אז דמות שאינה תלוית חומרה. למשל סמפור. חשוב להבין כי הסמפור עצמו ממומש בדרך כלל על ידי פסיקות, אולם לתהליך אין גישה ישירה לפסיקות אלו. לפיכך, שינוי במימוש הסמפור לא ישפיע כלל על התהליך עצמו.

הדרך הטובה ביותר להסתיר את הפסיקות היא שכאשר תהליך מתחיל פעולת קלט\פלט הוא נכנס למצב BLOCK עד אשר פעולת הקלט\פלט מסתיימת, ומתרחשת פסיקה. תהליך יכול להכניס את עצמו למצב חסום על ידי שימוש בפעולת סמפור לדוגמא, WAIT על משתנה תנאי או RECEIVE בהודעה. כאשר הפסיקה מסתיימת היא תבצע פעולה לשחרור התהליך שקרא לה. במערכות מסוימות יהיה זה על ידי ביצוע UP על סמפור, באחרות על ידי ביצוע סיגנל על משתנה במוניטור.

5.2.3 מתאמי התקנים – DEVICE DRIVER

מרכזים את כל חלקי התוכנה התלויים בהוראות חומרה במודול מיוחד – מתאם ההתקן. לכל סוג של התקן יש מתאם משלו. בצורה זו אפשר להגשים את עקרון האי-תלות של התוכנה. מבחינה מעשית – החלפת דיסק תגרור לכל היותר החלפה של מתאם התקן. בהנחה שהחלפת מתאם אכן נחוצה, מהו החלק המשותף לשני המתאמים?

הממשק המופשט שרואים רובדי התוכנה העליונים.

בתחילת הפרק ראינו שלכל בקר יש אוגר אחד או יותר שנותן לו פקודות. מתאמי ההתקנים מוציאים לפועל פקודות אלה ובודקים אם הפקודות בוצעו באופן מוצלח. אולם, מתאם הדיסק הוא החלק היחיד במערכת ההפעלה שיודע כמה אוגרים יש בבקר הדיסק, ולמה הם משמשים. הוא לבדו יודע על הסקטורים, הצילינדרים, הערוצים, הראשים, הזרוע, מתאם המנוע וכל שאר הפעולות המכניות שגורמים לעבודה תקינה של הדיסק. באופן כללי תפקידו של מתאם ההתקן הוא לקבל בקשות מופשטות מהתקן תוכנה עצמאי

שברמה מעליו ולראות האם היא ברת ביצוע. בקשה טיפוסית היא לקרוא בלוק N. אם המתאם לא עובד בשעה שהבקשה מגיעה הוא מתחיל לבצע אותה באופן מיידי, אולם, אם הוא עסוק עם בקשה, הוא יכניס בקשה חדשה לתוך תור של בקשות ממתינות לטיפול מוקדם ככל האפשר.

עמוד 213

השלב הראשון בביצוע בקשת קלט\פלט, נניח מהדיסק, היא לתרגם אותה מבקשה מופשטת למונחים קונקרטים. כלומר עבור מתאם הדיסק המשמעות היא הבנה היכן על גבי הדיסק נמצא הבלוק המבוקש, בדיקה לראות אם המנוע של המתאם עובד, האם הזרוע ממוקמת בצילנדר הנכון וכו'. כלומר המטרה היא להחליט איזה פעולות הבקר צריכות להתבצע ובאיזה סדר. משהחליט איזה פקודות עליו לשלוח לבקר, הביצוע מתחיל על ידי רישום הפעולות ברגיסטרים של הבקר. חלק מהבקרים יכולים לטפל בפקודה אחת בפעם אחת, בקרים אחרים מוכנים לקבל רשימה של פקודות שהם מבצעים באופן עצמאי ללא עזרה נוספת ממערכת ההפעלה. לאחר שהפקודות הוצאו, יכול לקרות מצב אחד מתוך שניים, במקרים רבים מתאם ההתקן חייב להמתין עד שהבקר עושה כמה פעולות, כך שהוא חוסם את עצמו עד שהפסיקה משחררת אותו. במקרים אחרים, הפעולות מסתיימות ללא דיחוי כך שהמתאם לא נחסם, לדוגמא, גלילת מסך בחלק מהמסופים מצריכה כתיבה של מספר בתים לתוך הרגיסטרים של הבקר, אין צורך בשום פעולה מכנית, כך שכל הפעולה יכולה להסתיים במספר מיקרו-שניות. במקרה הקודם המתאם החסום יתעורר על ידי הפסיקה, במקרה האחרון, הוא לעולם לא יישן. בכל מקרה אחרי שהפעולה הושלמה יש לבדוק שאין בה טעויות. אם הכל בוצע כשורה, למתאם יהיה נתונים להעביר לתוכנת קלט\פלט המשוחררת מאילוצי חומרה. לסיום הוא מחזיר מספר אינפורמציות הקשורות לדווח שגיאות בחזרה אל המתקשר אליו. אם ישנן בקשות נוספות בתור, אחת מהן תבחר ותתחיל לפעול, אם אין בקשות בתור, המתאם יחסם וימתין לבקשה הבאה.

5.2.4 <u>תוכנת קלט\פלט המשוחררת מאילוצי חומרה.</u>

למרות שחלק מתוכנות הקלט\פלט מתוכננות עבור התקנים ספציפים, חלקים גדולים מהם משוחררים מאילוצי חומרה. ההבדל המדויק בין המתאמים לתוכנה זו הוא תלוי מכונה, בגלל שכמה מהפונקציות שיכולות להתבצע על ידי התוכנה יכולות למעשה להיות מבוצעות על ידי המתאם. הפוקציות המוראות בתרשים 5-5 מבוצעות באופן טיפוסי בתוך הזאת. פרוט הפעולות שנעשות:

א. זיהוי הפעולות המשותפות לכל ההתקנים, ואספקת ממשק אחיד לרמת משתמש (תוכנה).

ב. חלק חשוב במערכת ההפעלה הוא איך אובייקטים כגון קבצים והתקני קלט\פלט נקראים. התוכנה מטפלת במיפוי שמות סמליים של התקנים כפי קרויים בתוך המערכת עצמה. לדוגמה ב- UNIX שם המתאם כגון DEV/TTY מציין באופן ייחודי את הי- I-NODE של אותו קובץ מסוים ואותו I-NODE מכיל את מספר ההתקן, אשר מונפק על מנת לאתר את המתאם המתאים. ה- I-NODE מכיל גם את מספר המתאם המשני אשר מועבר כפרמטר לתוך המתאם על מנת לציין את החלק שצריך לקרוא או לכתוב.

עמוד 214

ג. הגנה על ההתקן – כיצד המערכת תמנע גישה להתקנים מסוימים. בחלק מהמיקרו-מחשבים כגון MS-DOS אין הגנות בכלל, כל תהליך יכול לבצע ככל שירצה. ברוב מערכות MAINFRAM הגישה להתקני קלט\פלט על ידי משתמש אסורה בהחלט. ב-UNIX יש שיטה גמישה יותר, הקבצים הספציפים המטפלים בקלט\פלט מוגנים על ידי RWX ביטים. מנהל המערכת יכול לבחור הרשאות לכל התקן.

ד. לדיסקים שונים יש גודל סקטורים שונים. זה תפקידה של התוכנה להחביא את העובדה הזאת ולספק גודל בלוק אחיד עבור השכבות העליונות. לדוגמה, על ידי טיפול במספר סקטורים כבלוק לוגי אחד. בדרך זו השכבות הגבוהות יותר יתעסקו רק בהתקנים מופשטים המשתמשים כולם באותו גודל בלוק לוגי, בלי קשר לגודל הפיזי של הסקטור. באופן דומה, חלק מהתקני התווים שולחים את הנתונים שלהם בית אחד

בכל פעם (לדוגמא, קוראי סרט נייר), בעוד שהאחרים שולחים את הנתונים שלהם בחלקים גדולים, (לדוגמא, קוראי כרטיסים) הבדלים אלה חייבים להיות מוסתרים גם כן.

ה. מאגר נתונים – עבור בלוק או התקני תווים. עבור התקני בלוק החומרה מתעקשת לקרוא ולכתוב בלוקים שלמים בפעם אחת, אבל תהליכי המשתמש חופשיים לקרוא ולכתוב בחלקים שרירותיים. אם תהליכי המשתמש נכתבו על חצי בלוק, מערכת ההפעלה בדרך כל תשמור את הנתונים באופן פנימי עד שהמשתמש יסיים לספק בלוק שלם. ואז הבלוק ייכתב לדיסק. עבור התקני תווים, המשתמש יכול לכתוב נתונים למערכת מהר יותר מאשר היא יכולה לקבל, ולכן הנתונים נאגרים עד אשר הם ניתנים לטיפול.

ו. כאשר נוצר קובץ והוא מתמלא בנתונים חייבים להקצות בלוקים מהדיסק, עבור אותו קובץ. עבור פעולת ההקצאה, מערכת ההפעלה זקוקה לרשימה או מפת ביטים של הבלוקים הפנויים שבדיסק, אבל האלגוריתם להקצאת בלוקים פנויים לאחסון נתונים גם הוא משוחרר מאילוצי חומרה ויכול להתבצע מעל הרמה של מתאם ההתקן.

ז. חלק מההתקנים כגון מתאמי טייפים מגנטיים יכולים להשתמש רק בתהליך אחד בכל רגע נתון, ועל כן על מערכת ההפעלה לבחון את הבקשות עבור ההתקן שמופעל ולקבל או לדחות אותם לפי בחינה של המתאם הדרוש, אם הוא פנוי לעבודה או לא.
דרך פשוטה לטפל בבקשות אלו היא לדרוש תהליך המבצע פעולת OPEN על הקבצים הספציפיים עבור ההתקן ישירות.

עמוד 215

אם ההתקן איננו פנוי פעולת OPEN תיכשל, סגירה של התקן המוקדש לכך תשחרר אותו.

ח. הטיפול בשגיאות מתבצע על ידי המתאמים. רוב השגיאות הן קשורות להתקנים כך שרק המתאם יודע מה לעשות (לדוגמא, RETRY, IGNORE). שגיאה טיפוסית נגרמת על ידי בלוק פגום בדיסק שלא ניתן לקריאה יותר. לאחר שהמתאם ניסה לקרוא את הבלוק מספר פעמים הוא מתייאש ומודיע לתוכנה. כיצד להתייחס לשגיאה מכאן ואילך זוהי תוכנה המשוחררת מאילוצי חומרה. אם השגיאה ארעה כאשר קוראים קובץ של משתמש, יהיה מספיק לדווח על השגיאה בחזרה לקורא, ברם אם היא ארעה כאשר קוראים נתוני מערכת קריטי, כגון בלוק המכיל מפת ביטים המראים איזה בלוקים פנויים, למערכת ההפעלה לא תהיה ברירה אלא להדפיס הודעת שגיאה, ולסיים.
האם יש קשר בין בלוק לוגי לבלוק פיזי?
לא, אולם סביר להניח ששניהם יהיו בגדלים בינריים כגון 512K, 2048K וכדומה.

5.2.5 <u>תוכנת קלט\פלט ברמת תהליכי משתמש</u>

למרות שרוב תוכנות הקלט\פלט נמצאות בתוך מערכת ההפעלה, חלק קטן מהן המכיל ספריות המקושרות יחדיו עם תוכניות המשתמש, ואפילו תוכנית מלאה אשר רצה מחוץ לגרעין. קריאות מערכת המכילות קריאות ממערכת קלט\פלט נקראות בדרך כלל על ידי פרוצדורת ספריה. Count=write (fd,buffer,nbytes) ; פרוצדורת ספריה WRITE תקושר עם התוכנית ותיכלל בתוכנית הבינארית הנוכחית בזיכרון בזמן הריצה. האוסף של כל פרוצדורות הספריות הוא בברור חלק ממערכת הקלט\פלט. בעוד שפרוצדורות אלו עושות מעט יותר מאשר לשים את הפרמטרים שלהם במקומות המתאימים עבור קריאות המערכת, ישנן פרוצדורות קלט\פלט אחרות שעושות את העבודה האמיתית בפועל. דוגמא מ-C היא PRINTF אשר לוקחת מחרוזת ומשתנים אחרים כקלט, בונה מחרוזת אסקי, לאחר מכן קוראת WRITE אל מחרוזת הפלט. דוגמא של פרוצדורה דומה עבור קלט היא GETS אשר קוראת בשורה אחת ומחזירה אותה כמחרוזת. ספריות קלט\פלט הסטנדרטיות מכילות מספר פרוצדורות אשר מערבות קלט\פלט, וכולן רצות כחלק מתוכניות המשתמש.
ההבדל בין קריאת המערכת WRITE() לבין פוקציית הספריה PRINTF הוא שאחרונה בנויה מעל לראשונה וכוללת מנגנונים חיוניים נוספים כגון עיצוב (FORMAT) או חוצצים.

העיצוב הוא תוצאה של בקשת התהליך. השירות השני הוא שקוף וממלא למעשה תפקיד דומה לזה של מערכת ההפעלה, אך מתבצע ברובד תהליכי המשתמשים.
לרמה זו מצטרפים שדים למיניהם המשרתים את פעולות הקלט\פלט. למשל, שדים העוסקים בקביעת תור להתקנים ייעודיים (SPOOLERS), שדים השולחים ומקבלים דואר, שדים מפקחים, המודיעים על תקלות בהתקנים, ושדים הדואגים לתהליכים מורעבים שלא קבלו שירות זמן רב.
לא כל תוכנות קלט פלט ברמת המשתמש מכילות פרוצדורות ספריה. קטגוריה נוספת חשובה היא שיטת ה- SPOOLING. מערכת זו היא הדרך להתמודד עם משאבים בלעדיים במערכת בעלת ריבוי תהליכים העובדת במקביל. נתייחס להתקן להתקן ייעודי טיפוסי: המדפסת. אמנם טכנית פשוט יותר לתת לתהליך משתמש לפתוח את קובץ התווים המיוחדים עבור המדפסת, אולם נניח שתהליך יפתח את הקובץ, ולא יבצע דבר במשך שעות. אף תהליך אחר לא יוכל להדפיס דבר. במקום, יצרו תהליך מיוחד הנקרא שד, וספריה מיוחדת הנקראת ספריה ייעודית. כדי להדפיס קובץ, המשתמש שם תחילה את הקובץ בספריה הייעודית. רק לשד המדפסת יש רשות להשתמש בקובץ של המדפסת, ולהדפיס את הקבצים שנמצאים בספריה הייעודית. על ידי הגנת הקובץ המיוחד משימוש ישיר של משתמשים, הבעיה של משתמש שתופס את הקובץ שלא לצורך, נפתרת.
מערכת ה-SPOOLING אינה משמשת רק למדפסת, אלא גם למצבים אחרים.

עמוד 216

לדוגמא, מעבר קבצים ברשת כגון דואר אלקטרוני, משתמש בשד הרשת. כדי לשלוח קובץ למקום כלשהו המשתמש שם אותו בספרייה הייעודית של הרשת. מאוחר יותר שד הרשת מוציא את הקובץ ושולח אותו. רשת יכולה להיות מורכבת מאלפי מחשבים ברחבי העולם המתקשרים על ידי קווי טלפון ורשתות מחשבים. לשלוח דואר ב- USENET אתה קורא לתוכנית בשם SEND, אשר מקבלת מכתב לשליחה ואז מפקידה אותו בספרייה הייעודית, לשליחה מאוחרת יותר. כל מערכת הדואר רצה מחוץ למערכת ההפעלה.
שרטוט 6-5 מסכם את השכבות השונות לפי רמות המרכיבות את תוכנת ניהול ההתקנים המבצעים קלט\פלט.
כאשר תוכנית משתמש מנסה לקרוא בלוק מקובץ, לדוגמא, מערכת ההפעלה מבקשת להוציא לפועל קריאה. תוכנה משוחררת מאילוצי חומרה מחפשת התאמה. אם הבלוק המבוקש לא שם, היא קוראת למתאם התקן לאשר את הבקשה עבור החומרה. ואז התהליך נחסם עד שפעולת הדיסק מסתיימת. כאשר הדיסק מסיים, החומרה גורמת לפסיקה. מנהל הפסיקות מורץ לבדוק מה קרה. לפי הסטטוס בהתקן, הוא מעיר את התהליך הישן לסיים את בקשת הקלט\פלט כך שתהליך המשתמש יוכל לסיים

דיסקים

מימוש קלט\פלט בתוכנה –

מטרות תוכנת קלט\פלט:

1. עצמאות ההתקנים – קוד שלא מתחשב בסוג האמצעי.
2. שמות זהים
3. טיפול בשגיאות
4. טיפול במקרה שההתקנים אינם ניתנים לשיתוף כמו מדפסת

רבדי התוכנה:

1. מנהל הפסיקות
2. מתאם ההתקן
3. תוכנה המשוחררת מאילוצי חומרה
4. תוכנת המשתמש – תהליכים

מנהל הפסיקות:

הסיבות להסתרת הפסיקות:

א. הגנה מפני המשתמש

ב. שייך לחומרה – הסתרה
תפקיד מנהל הפסיקות:
כאשר מתחילה פעולת קלט\פלט התהליך נמצא במצב רדום עד אשר הפסיקה מסתיימת והפעולה הסתיימה, ואז התהליך מקבל את התוצאה לה חיכה.
מכניס את מתאם ההתקן לתרדמה ומעיר אותו.
תפקידי מתאם ההתקן:
א. מכיר את החומרה הספציפית לה הוא מותאם.
ב. מקבל בקשות מופשטות ממערכת ההפעלה (תוכנה משוחררת מאילוצי חומרה). ובודק אם היא ברת ביצוע.
ג. מתרגם בקשה מופשטת למונחים קונקרטיים – מיקום הבלוק המבוקש בדיסק, מיקום זרוע. ומחליט איזה פקודות לשלוח לרגיסטרים של הבקשה.
ד. נכנס למצב חסום (בדרך כלל) בעת פעולתו של הבקר (העברת נתונים).
ה. מתעורר על ידי פסיקה לאחר סיום פעולת הבקר ועוסק בחיפוש טעויות
ו. מעביר סטטוס פעולה לתוכנה המשוחררת מאילוצי חומרה.
תפקידי התוכנה המשוחררת מאילוצי חומרה:
אינו מותאם להתקן ספציפי ולכן תפקידיו כלליים יותר.
תוכנת הקלט\פלט ברמת התהליך:
א. חלק קטן מפוק' הספריה (שאינן מועברות לרמת גרעין)
ב. ספריה ייעודית. והתמודדות עם משאבים במקביל.

5.3 דיסקים

לדיסקים יש שלושה יתרונות מרכזיים על פני הזיכרון הראשי:
1. קיבולת אחסון גדולה יותר.
2. מחיר לבית נמוך יותר.
3. האינפורמציה נשמרת גם כאשר מכבים את המחשב.

רוב היתרונות נובעים ממחירו הזול של הדיסק לעומת הזיכרון.

עמוד 217

5.3.1 חומרת הדיסק

דיסק אופייני מורכב מצילינדרים, כאשר כל אחד מהם מחולק לגזרות – סקטורים. כל הסקטורים מכילים אותה כמות של נתונים, אם כי סקטורים הקרובים לחלק החיצוני של הדיסק ארוכים יותר, באופן פיזי מאלו הפנימיים. המקום הנוסף לא מנוצל.
הבקר יכול לנהל יותר מדיסק אחד. דבר זה נקרא חיפוש במקביל – OVERLAPPED SEEK. כאשר יש רק דיסק אחד, הדבר אינו אפשרי שכן כאמור כל הראשים בדיסק נעים יחד ובמקביל.
בקרים רבים יכולים לקרוא ולכתוב על דיסק אחד, בעוד הם מחפשים בדיסק נוסף אחד או יותר, אולם הם אינם יכולים לקרוא ולכתוב בו זמנית בכמה דיסקים. היכולת לחפש בכמה דיסקים במקביל מפחיתה את זמן הגישה הממוצע.

5.3.2 אלגוריתם לתזמון זרוע הדיסק

בקשות לדיסק מגיעות לבקר כדרישות לקריאה או כתיבה של בלוקים מסוימים. כדי למלא בקשות אלו יש לבצע את סדרת הפעולות הבאה:
א. על הראש להתייצב מעל המסלול המתאים. הזמן הדרוש לשם כך נקרא זמן החיפוש (SEEK TIME).
ב. על הדיסק להסתובב עד שהראש יימצא מעל לגזרה המתאימה. הזמן הדרוש לשם כך נקרא זמן הסיבוב (ROTATIONAL DELAY).
ג. עתה מתבצעת העברת הנתונים עצמה. הזמן הדרוש לכך נקבע על ידי קצב ההעברה. מרווח זמן זה נקרא זמן העברה (TRANSFER TIME).

המרכיב הראשון הוא המרכיב המשמעותי ביותר מבחינתה של מערכת ההפעלה כי שני המרכיבים האחרים נתונים על ידי החומרה ואין דרך מיידית לשפרם. לעומת זאת, בידינו לשפר באופן משמעותי את מספר התנועות הדרוש לראש כדי לספק קבוצת בקשות.
לפנינו שלושה אלגוריתמים מרכזיים לתזמון הזרוע:

<u>אלגוריתם התור FCFS</u>

הבקר מקבל בקשות בזו אחר זו ומבצע אותן לפי הסדר, FIRST COME, FIRST SERVED, במצב זה מעט ניתן לעשות על מנת לשפר את זמן החיפוש. ניתן לשפר את האלגוריתם על ידי כך שמחזיקים טבלת בקשות, שהאינדקס שלה הוא מספר הצילינדר, כך שכל הבקשות לאותו צילינדר ממוקמות יחדיו ברשימה מקושרת, שראשה הוא הכניסה הרלוונטית בטבלה.

אלגוריתם SSF, (SHORTEST SEEK FIRST): מטפל בבקשה הקרובה ביותר למיקום שהוא נמצא בו, על מנת למזער את זמן החיפוש.
נדגים את פעולת האלגוריתם: נניח דיסק עם 40 צילינדרים. נכנסת בקשה לקריאת בלוק בצילינדר 11. בעוד תהליך חיפוש הצילינדר מתבצע מגיעות בקשות נוספות לצילינדרים 12, 1,36,16,34,9 בסדר הזה. הבקשות נכנסות לטבלת הבקשות, לרשימה מקושרת נפרדת עבור כל צילינדר. (איור 7-5).
עם סיום ביצוע הבקשה הנוכחית (לצילינדר 11), הבקר מחליט באיזה בקשה לטפל עכשיו. לפי אלגוריתם התור הצילינדר הבא יהיה 1, אחר כך 36 וכן הלאה. האלגוריתם ידרוש תנועות זרוע של 10, 35,20,18,25,3 בהתאמה, סך הכל 111 צילינדרים. אולם הוא יכול לטפל בבקשה הקרובה ביותר למיקום שהוא נמצא בו, למזער את זמן החיפוש, למשל הסדר הבא 12,9,16,1,34,36. לפי סדר זה סך התנועות הזרוע הוא 61 צילינדרים בלבד.
אלגוריתם זה מקטין כמעט בחצי את תנועות הזרוע, יחסית ל- FCFS.

<u>*עמוד 218*</u>

הבעיה באלגוריתם זה היא הרעבה. הזרוע תיטה באיזהו שלב להיות בסביבות המרכז, כך הצילינדרים הנמוכים או הגבוהים יסבלו לרוב מהרעבה.
אלגוריתם המעלית:
אלגוריתם זה מנסה לפתור את הקונפליקט בין זמן חיפוש להרעבה. לפיו הבקר ממלא את הבקשות בכיוון מסוים כל עוד מגיעות בקשות בכיוון זה, אחרת הוא משנה כיוון. (בדומה למעלית, כאשר היא עולה, יכולים להצטרף כל הנוסעים כלפי מעלה. כאשר אין בקשות לעלות, ויש בקשות לרדת, היא משנה כיוון). התוכנה שומרת ביט אחד הקובע את הכיוון הנוכחי, UP או DOWN. כאשר טיפול בבקשה מסוימת מסתיים, הבקר בודק את הביט, אם הוא UP אזי הבקשה הגבוהה יותר מטופלת, אם כזו הביט משתנה ל- DOWN. (איור 8-5).

<u>עמוד 219</u>

בדרך כלל זמן החיפוש באלגוריתם זה גבוה מזמן החיפוש באלגוריתם SSF. תכונה חיובית שלו היא שעבור אוסף של בקשות, הרף הגבוה ביותר של סך תנועות הזרוע, הוא קבוע: והוא לכל היותר כפליים ממספר הצילינדרים.
מבין שלושת האלגוריתמים שנדונו, אלגוריתם התור הוא הבטוח ביותר למניעת הרעבה. באלגוריתם זה, זמן ההמתנה של בקשה הוא תמיד יחסי למיקומה בתור. שני האלגוריתמים האחרים משנים בהתמדה את העדיפויות למילוי בקשות בהתאם למרחקו של הבלוק המבוקש ממקומה הנוכחי של זרוע הדיסק. לכן ייתכן שבקשה מסוימת תידחה לזמן רב והתהליך המבקש יורעב.
אלגוריתם הסריקה:
אלגוריתם זה מקטין את סכנת ההרעבה של אלגוריתם המעלית, הנוטה להעדיף את המסלולים המרכזיים בדיסק. אלגוריתם הסריקה דומה מאוד לאלגוריתם המעלית, אלא שבמקום לעלות ולרדת בהתאם לדרישות הנוסעים, המעלית עולה בהתמדה תוך מילוי הבקשות בכיוון זה. כאשר המעלית מגיעה לקומה העליונה, היא חוזרת מייד (ריקה) לקומה הנמוכה ביותר שבה מצפים לה. בבניין שבו מותקנת מעלית מעין זו האנשים תמיד נאלצים לרדת ברגל, אולם הם יכולים להתנחם בעובדה כי כאשר יבקשו לעלות במעלית, הם ימתינו לה זמן קצר בהרבה (מאשר באלגוריתם הקודם).
כמה מפקחי דיסק מספקים דרך עבור התוכנה לבדיקת מספר הסקטור הנוכחי שמתחת לראש, לשם ייעול. אם ישנן 2 או יותר בקשות לאותו צילינדר, הבקר יכול לאשר בקשות

לסקטורים הבאים בתור שיעברו מתחת לראש. כאשר יש תורים מקביליים, ניתן לטפל רצף של בקשות בתורים שונים ללא עלות. לניצול זה של הראש אין השפעה מבחינת תנועות זרוע, וזמן סיבוב.
כאשר ישנם כמה בקרי דיסק, יש לשמור טבלת בקשות ממתינות עבור כל בקר.
בתחילת טיפול בבקשה מסוימת, ניתן לבדוק אם אחד מהבקרים ממוקם מעל הצילינדר הדרוש, ואז הבקשה תועבר אליו. אם לא, אזי על הבקר לטפל בהגעה לצילינדר הדרוש בעצמו.

ניתן לשפר על ידי טכנולוגיה את זמן החיפוש, אולם זמן הסיבוב לא משתנה. יש דיסקים שזמן החיפוש הממוצע קטן מזמן הסיבוב. עם מגמה זו תמשך זמן הסיבוב יהיה דומיננטי, ויהיה הכרחי לשנות את האלגוריתמים, כך שיטפלו בבעיה זו.
מערכות מחשב רבות מכילות יותר מדיסק אחד. במצב זה ניתן להגביר הן את מהירות הגישה (על ידי ניהול תורים מקביליים) והן את אמינות המערכת (על ידי גיבוי הדדי).
מנגנון מעניין הוא מערך הדיסקים (RAID). השיטה משלבת העברה במקביל (ולפיכך חיסכון גדול בזמן העברה) ובטיחות נתונים הנוצרת באמצעות הוספת סיביות של קוד לתיקון שגיאות. סיביות אלו מאפשרות לשחזר את הנתונים גם כאשר אחד הדיסקים במערך קורס.

עמוד 220

5.3.3 <u>טיפול בשגיאות</u>

השגיאות השכיחות הן:

1. שגיאות תכנות. (פניה לסקטורים שלא קיימים). רוב הבקרים בודקים אם הפרמטרים שהם מקבלים חוקיים, ומדווחים אם לא. בתאוריה שגיאה זו לא צריכה להתרחש אך כאשר הבקר מאותת שתקלה כזו התרחשה, המתאם צריך לעצור ולהדפיס הודעה או לא לבצע את הבקשה.
2. שגיאות בבדיקות ארעיות. (נגרמות בגלל אבק על הראש). האבק נכנס באוויר בין ראש הדיסק והמשטח של הדיסק. ברוב הפעמים ניתן להתגבר על כך על ידי חזרה על הפעולה מספר פעמים. אם הטעות חוזרת על הצמה, הבלוק יכול להיות מסומן כמקולקל BAD BLOCK. דרך אלטרנטיבית היא כאשר הבקר שומר מספר מסילות שאינן לשימוש המשתמש, וכך כאשר בלוק מסוים מתקלקל, הוא מוחלף בבלוק השמור. הטבלה שממפה מסילות מקולקלות, למסילות רזרביות נשמרת בזיכרון הפנימי של בקר הדיסק. עקב כך יורעו ביצועי אלגוריתם המעלית אם נחליף את בלוק 3 בבלוק 800 לדוגמא.
3. שגיאות בבדיקות קבועות (בלוק בדיסק ניזוק פיזית).

עמוד 221

4. שגיאות חיפוש (הזרוע נשלחת לצילינדר 6 אולם מגיעה לצילינדר 7). הבעיה נגרמת כתוצאה מבעיות מכניות בזרוע. הבקר עוקב אחר תנועות הזרוע באופן פנימי, כאשר חלק מהבקרים מתקנים את הטעויות. אחרים רק מאתחלים את סיבית הטעות ומשאירים את שאר העבודה למתאם ההתקן הקורא לפקודה RECALIBRATE שתזיז את הזרוע מהמקום בו היא נמצאת ותאתחל את מיקום הצילינדרים ל- 0. בעיה זו נפתרת בדרך כלל, אם לא יש לתקן את מתאם ההתקן (DRIVER).
5. שגיאות הבקר (הבקר מסרב לקבל פקודות). הבקר הוא מחשב זעיר, עם תוכנה, משתנים, מחסנית, וגם באגים. לעיתים צרוף מקרים יגרום לבקר להיכנס ללולאה, או לשגיאת תוכנה אחרת. לשם כך יש שבב, שכאשר הוא "נדלק" הבקר מאפס את עצמו. אחראי על כך הוא בקר ההתקן שיש לו סיבית איפוס שאחראית על איפוס הבקר. אם זה לא עוזר יש להוציא הודעת שגיאה.

5.3.4 <u>זיכרון מטמון</u>

כאשר ראש קורא מתייצב מעל מסלול מסוים, כל הראשים האחרים נמצאים מעל מסלול זה במשטחים אחרים. קריאה מכל הראשים במקביל היא פעולה זולה, שכן איננה דורשת תנועה פיסית. מידע זה ניתן לאחסון ולשמירה למקרה שיידרש בהמשך. הרעיון הוא לאגור

את המידע בתוך חוצץ ולהשתמש בו במידת הצורך. פעולה זו יכולה להיעשות בתוכנה, באמצעות המתאם, או בצורה טובה יותר – על ידי הבקר עצמו.
במתאמים הנוכחיים זמן החיפוש אחר צילינדר חדש הוא עדיין ארוך יותר מזמן הסיבוב, או זמן העברת הנתונים. כאשר הזרוע כבר זזה אין זה משנה הרבה אם כל הסקטור ייקרא, או כל המסילה. ישנם מתאמי התקנים שמחזיקים זיכרון מטמון פנימי שמוחבא מרמת התוכנה המשוחררת מאילוצי חומרה. המידע נאגר בחוצץ זה, ולכן אם בעתיד יצטרכו סקטור מסוים שנמצא בו לא יהיה צורך בחיפוש והעברה מהדיסק.
החיסרון הוא – בנוסף לכך שהתוכנה מסובכת יותר וצריך מקום עבור החוצץ, מעבר הנתונים מהזיכרון מטמון הפנימי לתוכנית הקוראת יצטרך להיעשות על ידי ה- CPU בלולאה בתוכנית ולא על ידי החומרה - DMA.
שיפור יכול להיעשות כאשר הבקר עצמו מכיל זיכרון מטמון פנימי המועבר למתאם כך שהתעבורה בין הבקר והזיכרון יכולה להיעשות על ידי ה- DMA ואין טעם שגם מתאם ההתקן יעשה זאת.

עמוד 222

5.3.5 דיסק בזיכרון – RAM DISKS

הרעיון הוא פשוט, שימוש בחלק שמוקצה מראש בזיכרון הראשי עבור אחסון בלוקים עם מידע, עם שתי פקודות: כתוב בלוק, קרא בלוק. בדרך כלל בלוקים אלו מאוחסנים על דיסקטים או על דיסק קשיח.
לדיסק בזיכרון יש יתרון שזמן הגישה הוא מיידי (אין זמן חיפוש או זמן סיבוב) ואין תקלות טכניות, כך שנוח לאחסן שם תוכניות או נתונים שנזקקים להם תכופות. איור 5-9 מראה RAM DISK. הוא מחולק ל- N בלוקים, לפי כמות הזיכרון שהוקצתה לו. כל בלוק הוא באותו גודל של הבלוקים שבדיסק. כאשר ניתנת למתאם הוראה לקרוא, או לכתוב בדיסק הוא מחשב היכן נמצא הבלוק בדיסק שבזיכרון וכותב או קורא בו. הדבר מבוצע לרוב על ידי פרוצדורה באסמבלי.
החיסרון העיקרי הוא המחיר.

5.4 שעונים

שעונים הם מרכיב חיוני בכל מערכת מחשב, ובפרט במערכת רבת תהליכים. תפקידים נוספים לשעון, מלבד אלו שיפורטו למטה הם:

א. תאום בין תהליכים.
ב. קביעת גילם של תהליכים וקבצים.
ג. הפעלה אוטומטית של תהליכים התלויים בזמן או שצריכים להתעורר בזמן נתון.
ד. שומרים את השעה ביום, ומונעים מתהליך אחד שיהיה לו מונופול על ה- CPU.

תוכנת השעון דומה לזו של מתאם התקן, אם כי שעון הוא לא התקן בלוקים כמו דיסק, או התקן תווים כמו מסוף.

עמוד 223

5.4.1 חומרת השעון

השעון במחשב הוא מכשיר פשוט ביותר. הוא מכיל התקן פיסי הרוטט בקצב מהיר מאוד וקבוע, מונה ספרתי ויחידה הניתנת לתכנות. יחידה זו קובעת מתי ייצר השעון פסיקות. כמעט תמיד צמוד ליחידה זו גם זיכרון עצמאי המאפשר לשמור או לעדכן את הזמן במחשב.
שני סוגי שעונים נפוצים במחשבים, והם שונים מהשעונים שבשימוש על ידי אנשים. השעון הפשוט יותר קשור לזרם של 110 או 220 וולט, וגורם לפסיקה מחזוריות.
סוג נוסף של שעון בנוי משלושה רכיבים: מתנד קריסטל (מכשיר ליצירת זרמי חשמל משתנים), מונה ורגיסטר (ראה איור 10-5). השעון יוצר סיגנלים בתדירות גבוהה, תלוי במתנד. הסיגנל מועבר אל המונה, שמוקטן עד ל- 0. כאשר המונה מגיע ל- 0, הוא גורם לפסיקת CPU. מה שקורה לאחר מכן תלוי במערכת ההפעלה. לשעונים כמה אופני פעולה.
ONE-SHOT MODE - כאשר השעות מאותחל הוא מעתיק את הערך שנמצא בריגיסטר אל המונה, ואז בכל פולס של הקריסטל, הוא מפחית את המונה. כאשר מגיעים ל- 0 הוא גורם לפסיקה ועוצר עד שהוא מאותחל שוב על ידי התוכנה.

SQUARE-WAVE MODE – לאחר שהוא מגיע ל- 0 וגורם לפסיקה, תוכן הריגיסטר מועתק באופן אוטומטי אל המונה, וכל התהליך מתחיל מחדש באופן עצמאי.
פסיקות מחזוריות אלו נקראות CLOCK TICKS.
היתרון בשעון מתוכנת הוא שניתן לשלוט בפסיקות על ידי תוכנה. בקריסטל של 1 מגהרץ כל פולס יתרחש כל מיקרו-שניה. עם רגיסטר של 16 ביטים, פסיקה תתרחש בין 1 מיקרו-שניה, לבין msec65 . שבב השעון, מכיל שנים או יותר שעונים מתוכנתים שונים, ויש להם הרבה אפשרויות (ספירה למעלה, במקום למטה, ניתוק פסיקות ועוד).
כדי לבצע שעון יום התוכנה שואלת את המשתמש לזמן הנוכחי, ואז מתרגמת אותו למספר פעימות השעון מאז 1 ינואר 1970, כמו ב- UNIX. בכל פעימת שעון הזמן האמיתי עולה באחד. על מנת למנוע את אובדן השעה, היות ומכבים את המחשב, יש מחשבים שמאחסנים את הזמן האמיתי ברגיסטר מיוחד המופעל על ידי בטריה.

עמוד 224

5.4.2 תוכנת השעון

תפקידו של השעון הוא ליצור פסיקות בקצב קבוע מראש. התוכנה של מתאם השעון משתמשת בפסיקות אלה כדי לבצע את שש המטלות המתוארות בסעיף זה. המטלות של תוכנת השעון:

1. שמירה של הזמן ביום – שמירת הזמן מהראשון בינואר 1970. ישנה בעיה לשמור מספר זה ב- 32 סיביות. ישנן שלוש גישות לפתור את הבעיה:
   א. לשמור ב- 62 סיביות מה שיגרום שפעולת ההוספה למונה תהיה פעולה יקרה יותר היות והיא תבוצע כמה פעמים בשניה.
   ב. שמירת הזמן בשניות, במקום בפעימות, תוך שימוש במונה נוסף לספירת הפעימות, עד ששניה שלמה תושלם. כאשר משתמשים ב- 32 ביטים תקרה גלישה בשנת 2038.
   ג. ספירת הפעימות באופן יחסי לפעם האחרונה שהמערכת אותחלה. כאשר משתמש מכניס את השעה אל המחשב, זמן האיתחול מחושב ומאוחסן בזיכרון. אחר כך כאשר יש בקשה לזמן, מוספים את המונה, אל הזמן שאוחסן, על מנת לקבל את הזמן הנוכחי. כל שלושת הגישות משורטטות באיור 11-5.
2. מניעת מצב בו תהליכים ירוצו יותר מידי זמן - בכל פעם שתהליך מתחיל לפעול, המתזמן מאתחל את המונה, לערך של קצובת הזמן של התהליך בפעימות שעון. בכל פסיקת שעון, מתאם השעון מקטין את מונה הזמן של התהליך ב- 1. כאשר מגיעים ל- 0, המתאם קורא למתזמן שיפעיל תהליך נוסף.
3. בחינת ניצול המעבד – בחינה זו נחוצה הן לצרכים פנימיים של המערכת, והן לצורכי המשתמשים. לצרכים של ניהול מערכת ההפעלה, יעילות הניצול של המעבד היא מבחן שעל פיו ניתן לקבוע אם יש להגדיל או להקטין את דרגת ריבוי התוכניות, וזהו גם מבחן לקביעת העדיפויות לתזמון תהליכים. המתכנת, למשל, זקוק למדידת הזמן הנדרש להרצת קטעי קוד שונים כדי לייעל את פעולתם וכדי לגלות בהם שגיאות סמויות.

עמוד 225

ניתן ליישם זאת על ידי החזקת שעון נוסף שכאשר תהליך מופסק ניתן לדעת כמה זמן הוא רץ. ערך השעון צריך להישמר בכל פעם שמתרחשת פסיקה.
דרך אחרת היא להחזיק מצביע לטבלת התהליכים לתהליך הנוכחי שרץ במשתנה גלובלי. בדרך זו בכל פעימת שעון יש שדה בכניסה של התהליך הנוכחי שמוגדל. בצורה זו התהליך "מחויב" בכל פעימת מונה. החיסרון הוא שפסיקות מתרחשות בזמן ריצת תהליך, והוא ממשיך להיות מחויב גם בזמן הפסיקה, למרות שלא הספיק לפעול הרבה.

4. טיפול בקריאות מערכת ALARM שנוצרות על ידי תהליכי משתמש. ב- UNIX ובמערכות נוספות תהליך יכול לבקש ממערכת ההפעלה לתת אזהרה לאחר פרק זמן מסוים. יישום אחד הוא ברשת, כאשר אישור על נתונים שנשלחו, לא מגיע בתוך פרק זמן מסוים, אזי הנתונים נשלחים בשנית. יישום נוסף הוא הוראות עזר, כאשר סטודנט לא מספק תגובה למחשב בפרק זמן מסוים שנקבע לתשובה.
5. אספקת שעונים לשמירה, עבור חלקים מהמערכת עצמה. – למשל, על מנת להשתמש בדיסקט, על המערכת להפעיל את הכונן ולחכות כ- 500 מילי-שניות עד

שהוא יגיע למהירות הרצויה. לאחר סיום פעולת קלט\פלט ניתן להשתמש בשעון על מנת לכבות את המנוע, אם פעולה נוספת לא נדרשת. דוגמא נוספת היא שמסופים יכולים להדפיס 200 תווים בשניה, אולם הראש המדפיס יכול לחזור לשמאל תוך 5 מילי-שניות בלבד, ולכן על מתאם המסוף, ליצור השהיה לאחר תו סוף שורה.

המכניזם של השעונים לשמירה (WATCHDOG) דומה לסיגנלים של משתמש. השוני הוא שבמקום ליצור סיגנל, מתאם השעון קורא לפרוצדורה המסופקת על ידי המזמין, והיא חלק מקוד קריאה לשעון השמירה. הפרוצדורה יכולה לבצע הכל, כולל פסיקה.

6. אפיון פעולתם של תהליכים (PROFILING) הוא משימה חשובה ביותר לצורכי פיתוח תוכנה. האפיון מאפשר למדוד מהי צריכת הזמן של כל קטע קוד. לעתים קרובות מתגלות בדרך זו שגיאות שלא ניתן היה לגלות בכל דרך אחרת. ברור גם כי לקטעי תכנית הצורכים זמן מעבד מקסימלי יש לכתוב קוד אופטימלי.

ממש כשם שמערכת ההפעלה מדמה את פעולתם של מעבדים רבים במקביל על ידי ריבוי תהליכים, כך תוכנת השעון מדמה את פעולתם של שעונים רבים ככל שנדרש על ידי התהליכים. דרך אחת היא לשמור טבלת זמנים עבור זמני סיגנלים. כאשר הזמן הנוכחי מעודכן, המתאם בודק אם הסיגנל הקרוב ביותר התרחש. אם כן הטבלה נסרקת למציאת הסיגנל הבא שיתרחש.

דרך נוספת היא ייצוג של שעונים אלו ברשימה ממוינת, דבר המפשט מאוד מנגנון זה (ראה איור 12-5). בכל צומת ברשימה רשום כמה פעימות מונה, יחסית לצומת הקודם, יש לחכות לפני שגורמים לסיגנל.

עמוד 226

נעיר שבמשך פסיקות השעון, על מתאם השעון לבצע כמה מטלות: הגדלת הזמן הנוכחי, הקטנה הזמן המוקצב לתהליך (QUANTUM), ובדיקה האם הוא 0, הקטנת מונה האזעקה. כל פעולה כזו מהירה מאוד היות והיא מתבצעת הרבה פעמים בשניה.

פרק 6 : קיפאון – DEADLOCKS

עמוד 240

משאבים רבים במחשב אינם ניתנים לחלוקה בו-זמנית. משאבים אלו נקראים משאבים בלעדיים.

דוגמה למשאבים הניתנים לחלוקה :

א. זיכרון. עקרונית תהליכים מספר יכולים לפנות בו זמנית למקומות שונים בזיכרון.

ב. מסך. גם הוא יכול להתחלק בו-זמנית (למשל כמה חלונות).

ג. קריאת מצב השעון (לא עדכונו!).

תופעת הקיפאון קשורה לשימוש במקביל של תהליכים שונים במשאבים בלעדיים. כשאנו מדברים על משאבים, אין אלו בהכרח משאבים פיסיים. גם משאבים לוגיים כמו משתנים של תכנית או רשומה בקובץ הם משאבים, וככאלה גם הם לעתים קרובות בלעדיים.

דוגמא מובהקת למשאב לוגי בלעדי היא קטעים הקריטיים. תופעת הקיפאון בצורתה הפשוטה ביותר מתרחשת כאשר קיימים שני תהליכים, שכל אחד מהם מחזיק ברשותו משאב בלעדי הנחוץ לתהליך האחר. במצב זה אף אחד משני התהליכים אינו יכול להתקדם, והם נמצאים בקיפאון.

מערכות מחשב כוללות משאבים רבים שאינם ניתנים לשימוש בו-זמנית על ידי מספר תהליכים, כגון מדפסות, מתאם ההתקן של טייפ, אסימונים בטבלת I-NODE וכו'. אם שני תהליכים יכתבו בו זמנית במדפסת נקבל זבל. אם שני תהליכים ישתמשו באותו אסימון בטבלת I-NODE אזי מערכת הקבצים תשתבש. אי לכך לרוב מערכות ההפעלה יש יכולת לאפשר לתהליך מסוים שימוש בלעדי במשאב, או כל המשאבים שהוא דורש לפעילות מסוימת. למשל תהליך המעתיק קובץ הגדול מהדיסק מסרט מגנטי אל המדפסת זקוק לשימוש בלעדי הן במדפסת והן במתאם ההתקן בו-זמנית. במערכת עם תהליך אחד בלבד, התהליך יכול לקבל בפשטות כניסה לכל המשאבים להם הוא זקוק.

אולם במערכת רבת תהליכים יכולים להתעורר בעיות חמורות. נניח לדוגמא שני תהליכים שכל אחד מהם רוצה להדפיס קובץ גדול מאוד מטייפ. תהליך A מבקש רשות להשתמש במדפסת ומקבל אותה. תהליך B מבקש רשות להשתמש במתאם ההתקן וגם הוא מקבל אותה. כעת A מבקש את הטייפ, אולם בקשתו נדחית עד ש- B ישחרר אותו. לדאבוננו, במקום לשחרר את הטייפ, B מבקש את המדפסת. בשלב זה שני התהליכים חסומים לעד. מצב זה נקרא DEADLOCK קיפאון.

קיפאון יכול להתרחש במצבים רבים מלבד בקשה להתקן קלט\פלט בלעדי. במערכות בסיסי נתונים, תוכנית יכולה לנעות כמה רשומות שבהן היא משתמשת, כדי להימנע ממצב תחרות (RACE CONDITIONS). אם תהליך A נועל רשומה R1 ותהליך B נועל רשומה R2, ואז כל תהליך מנסה לנעול את הרשומה של השני, גם כאן נקבל קיפאון. לפיכך קיפאון יכול גם להתרחש כשהמשאבים הם לוגיים.

מצב קיפאון – מצב בו קיימים שני תהליכים או יותר, שכל אחד מהם מחזיק ברשותו משאב בלעדי הנחוץ לתהליך אחר. במצב זה אף אחד משני התהליכים אינו יכול להתקדם, והם נמצאים בקיפאון.

עמוד 241

6.1 משאבים

לצורך הדיון בנושא הקיפאון, נניח כי כאשר אנו מדברים על משאבים, הכוונה היא למשאבים בלעדיים, אלא אם כן נציין במפורש אחרת.

משאב : - חומרה כגון מדפסת או טייפ.

- מידע כגון משתנים, קטע קריטי.

משאב הוא כל דבר שניתן לשימוש על ידי תהליך בודד בזמן נתון.

אנו נבחין בין משאבים הניתנים לחילוץ (PREEMTABLE) לבין משאבים שחילוצם הכפוי מידיו של תהליך יגרום נזק משמעותי (ולעתים בלתי הפיך) לתהליך זה. רק דרישה למשאבים מהסוג השני עלולה לגרום לקיפאון. משאבים מהסוג הראשון ניתנים לחילוץ

ללא קושי, ובדרך זו גם קל להיחלץ ממצב שאחרת היה מוביל בוודאות לקיפאון. תהליך הדורש משאב עשוי לקבלו ואז להמשיך, או להמתין לו ולישון עד שיקבלו. מכל מקום, ההנחה היא כי תהליך איננו יכול להמשיך כל עוד הוא לא קיבל משאב שביקש. לאחר השימוש, המשאבים מוחזרים למערכת ההפעלה, אם ביוזמת התהליך ואם בעקבות סיומו.
ישנם שני סוגי משאבים:

1. משאבים הניתנים לחילוץ – PREEMPTABLE RESOURCE. הם משאבים שניתן לקחת אותם מהתהליך שהם נמצאים בבעלותו והדבר לא יגרום לנזק, ואף ימנע מצב של קיפאון שהיה נגרם בוודאות לו המשאבים לא היו מסוג זה. לדוגמא זיכרון. לדוגמא, מערכת עם 512K של זיכרון משתמש, מדפסת אחת ושני תהליכים בגודל 512K שכל אחד מהם רוצה את המדפסת. תהליך A מקבל את המדפסת, ואז מתחיל לחשב את הערך שהוא רוצה להדפיס. לפני שהוא מסיים עם החישובים, תם הזמן המוקצב לו והוא מוחלף. תהליך B מנסה להשיג את המדפסת גם כן. והגענו למצב קיפאון, כי ל- A יש את המדפסת, ול-B יש את הזיכרון, ואף אחד מהם לא יכול להמשיך מבלי המשאב שמוחזק על ידי האחר. למרבה המזל, ניתן לחלץ את הזיכרון מתהליך B על ידי החלפה עם A. כעת A יכול לפעול, להדפיס ואז לשחרר את המדפסת. הקיפאון נמנע.
2. משאב שלא ניתן לחילוץ (NONPREEMPTABLE) - משאבים שחילוצם הכפוי מידיו של תהליך יגרום נזק משמעותי (ולעיתים בלתי הפיך) לתהליך זה. (לדוגמא, מדפסת שתהליך החל כבר להדפיס בה).

באופן כללי, בקיפאון מעורבים משאבים מסוג שני. קיפאון פוטנציאלי שמעורבים בו משאבים הניתנים לחילוץ, ניתן לפתרון על ידי העברת המשאב לתהליך השני.
רצף האירועים בדרישה לשימוש במשאב:

1. בקשה לשימוש במשאב.
2. שימוש במשאב.
3. שחרור המשאב.

אם המשאב אינו פנוי בשלב הבקשה, התהליך מחויב להמתין. במספר מערכות הפעלה, התהליך נכנס באופן אוטומטי למצב חסום, ומתעורר כאשר המשאב נהייה זמין.

עמוד 242

במערכות אחרות הבקשה נכשלת עם ERROR CODE, והתהליך המבקש יכול להחליט להמתין מעט, ואז לנסות שוב.
תהליך שבקשתו למשאב נדחתה יכנס ללולאה המבקשת את המשאב, יישן, וינסה שוב. אמנם התהליך לא חסום לכל דבר, אולם הוא נקרא חסום היות והוא אינו יכול לבצע כל עבודה מועילה. לכן נניח בהמשך שאם בקשת תהליך נדחית, אזי הוא הולך לישון.
אופן בקשת המשאבים היא תלויה במערכת. ישנן מערכות שמשתמשות בקריאת מערכת REQUEST כדי לאפשר לתהליכים לבקש משאבים בלעדיים. ובמערכות אחרות הבקשות הן דרך קבצים מיוחדים שרק תהליך אחד יכול לגשת אליהם בו זמנית, והם נפתחים על ידי קריאת המערכת OPEN.

6.2 קיפאון DEADLOCK

הגדרה: קבוצת תהליכים נתונה במצב קיפאון, כאשר כל תהליך בקבוצה מצפה לאירוע הנמצא בשליטתו הבלעדית של תהליך אחר בקבוצה. שליטה בלעדית פירושה כי תהליך המצפה לאירוע אינו יכול להשפיע בדרך כלשהי על התרחשותו.
דוגמא לאירוע שטרם התממש אך היעדרו עלול להיות בין הגורמים לקיפאון:
למשל, תהליך עשוי לחכות לסיומו של תהליך אחר. המשאב הוא סיום התהליך. נניח כי תהליך B יכול להתחיל לפעול רק לאחר שתהליך A הסתיים, אבל כדי לסיים, תהליך Aזקוק למשאב המוחזק על ידי תהליך B. שני התהליכים נמצאים במצב של קיפאון!

6.2.1 תנאים לקיפאון

1. מניעה הדדית (MUTUAL EXCLUSION) – כל משאב מוקצה לתהליך אחד לכל היותר או שהוא חופשי.

2. החזקה והמתנה (HOLD AND WAIT CONDITION) – תהליך המחזיק משאב יכול לדרוש משאב אחר בלי לשחרר את המשאב שהוא מחזיק.
3. אין חילוץ כפוי – רק התהליך המחזיק במשאב יכול לשחררו.
4. מעגל ההמתנה (CIRCULAR WAIT) - כדי שיתקיים קיפאון בקבוצת תהליכים, חייב להתקיים מעגל שבו כל תהליך מחזיק במשאב הנדרש על ידי התהליך הבא במעגל.

כל ארבעת התנאים צריכים להתקיים כדי שיתרחש קיפאון. אם אחד מהם יחסר, לא יתרחש קיפאון.

עמוד 243

6.2.2 מודל קיפאון

כאשר קיים רק משאב אחד מכל סוג, ניתן לייצג את ארבעת התנאים על ידי גרף מכוון. בגרף זה יש רק שני סוגים של צמתים: צמתים המייצגים את המשאבים וצמתים המייצגים את התהליכים. הגרף בנוי כך שכל קשת מצומת תהליך אל צומת משאב מייצגת בקשה של התהליך לקבל את המשאב, וכל קשת ממשאב אל תהליך מצביעה על כך שהמשאב נמצא בחזקתו של התהליך. אנו יודעים כי תהליך המבקש משאב נמצא במצב של המתנה. ראה שרטוט 6-1.

כיצד מתבטא כל אחד מהתנאים לקיפאון במודל גרף הקצאת המשאבים?

1. מצומת משאב בגרף יוצאת קשת אחת לכל היותר.
2. מצומת תהליך יכולה לצאת קשת אל משאב גם בלי לנתק את הקשת המגיעה אליו ממשאב אחר.
3. קשת ממשאב אל תהליך יכולה להיות מנותקת רק על ידי התהליך עצמו.
4. מעגל קשתות בגרף הקצאת המשאבים מצביע על קיומו של קיפאון.

נציג דוגמא לשימוש בגרף. נניח שלושה תהליכים A,B,C ושלושה משאבים R,S,T. הבקשות והשחרור של שלושת התהליכים מודגמות באיור 6-2. מערכת ההפעלה חופשייה להריץ כל תהליך לא חסום בכל רגע, כך שהיא יכולה להריץ אותם באופן סידרתי אם תרצה (כל תהליך יפעל עד שיסיים).

סדר ריצה זה לא יגרום לקיפאון (כי אין תחרות על משאבים) אולם אין כאן מקביליות. בתוספת לבקשה ושחרור של משאב, תהליך מחשב ומבצע פעולות קלט\פלט. כאשר תהליכים רצים באופן סידרתי, אין אפשרות שכאשר תהליך אחד ימתין לאמצעי קלט\פלט, תהליך אחר ישתמש ב- CPU. לפיכך הרצת תהליכים באופן סדרתי, אינה אפשרות שנשתמש בה. מאידך, אם אף אחד מהתהליכים לא זקוק לקלט\פלט, הגישה של SHORTEST JOB FIRST עדיפה על ROUND ROBIN, כך שבתנאים מסוימים הרצת תהליכים באופן סידרתי יכולה להיות עדיפה.

עמוד 244

נניח כעת שכל התהליכים גם מחשבים וגם מבצעים קלט\פלט, כך שאלגוריתם ROUND ROBIN מתאים. הבקשות יכולות להתרחש בסדר שבאיור 6-2d. ואזי הגרף לששת הבקשות משורטט באיור 6-2e עד 6-2j. לאחר שבקשה 4 מתבצעת, A נחסם, ממתין ל- S, (איור 6-2h). בשני השלבים הבאים תהליכים B, C גם נחסמים.

עמוד 245

מערכת ההפעלה לא נדרשת לסדר מסוים בהרצת התהליכים, ולכן, אם אישור בקשה יוביל לקיפאון, מערכת ההפעלה יכולה להשעות את התהליך, מבלי לאשר את בקשתו, עד שזו תתאפשר. על ידי הרצת תהליכים A,C בלבד נקבל בקשות ושחרורים (איור 6-2k) במקום 6-2d. רצף זה יוביל לגרף 6-2q-l וימנע קיפאון. לאחר שלב q תהליך B יכול לקבל את S כי A ו C סיימו. אפילו אם B יחסם כאשר יבקש את T, לא יתרחש קיפאון. B רק ימתין עד אשר C יסיים.

יש קשר הדוק בין מקביליות לבין בעיית הקיפאון. ללא מקביליות אין קיפאון! הבחנה זו מזכירה את המצב לגבי בעיית הקטע הקריטי. גם שם הבעיה מתעוררת רק כאשר ישנה הרצה במקביל. עוד מסתבר (לפי הדוגמא שבאיור 6-2) כי בעיית הקיפאון קשורה קשר הדוק לסדר הקצאת המשאבים. כיוון שהקצאת המשאבים היא התפקיד של מערכת ההפעלה, ננסה לתקוף את הבעיה באמצעות ניסיונות לקביעת סדר ההקצאה הנכון.

באופן כללי ישנן ארבע אסטרטגיות להתמודדות עם בעיית הקיפאון:

א. התעלמות.
ב. גילוי והחלמה.
ג. התחמקות.
ד. מניעה.

6.3 אלגוריתם בת היענה

אלגוריתם זה, שנראה משונה במבט ראשון, נובע מהמחיר הגבוה שגובים הפתרונות למניעת קיפאון. פתרון זה מבוסס על העובדה הידועה כי במקרים מסוימים התרופה למחלה עלולה להזיק לחולה יותר מאשר המחלה עצמה. כפי שראינו, הקצאה בלתי מבוקרת של משאבים עלולה לגרום לקיפאון, והדרך היחידה למניעת הקיפאון היא פיקוח על החלוקה ועל דרך השימוש במשאבים. אך המחיר עלול להיות כבד מדי. פיקוח הדוק הופך בקלות למכשיר דיכוי. אם נזכור כי כל מה שתהליך עושה הוא שימוש במשאבים, הרי שפיקוח מעין זה מצריך זמן חישוב ניכר. חמור מזה: ההגבלות על חופש פעולתם של התהליכים עלולות להוות נטל כבד ביותר על המשתמשים.

דוגמא להגבלה מעין זו: כאשר לכל תהליך מותר להחזיק רק משאב אחד לכל היותר – לא ייתכן קיפאון. ברור כי הגבלה זו היא בלתי נסבלת לגבי רוב התהליכים.

הגישה הפשוטה ביותר היא אלגוריתם בת היענה: הטמן את הראש בחול, והעמד פנים כי הבעיה לא קיימת. עבור מתמטיקאים, לא מתקבל על הדעת שיש למנוע קיפאון בכל מחיר. מהנדסים ישאלו, מה התכיפות שצופים לבעיה, מה התכיפות שהמערכת מתמוטטת בגלל סיבות אחרות, וכמה חמור הוא קיפאון. אם קיפאון מתרחש אחת לחמש שנים בממוצע, אולם המערכת מתמוטטת עקב חומרה, שגיאות קומפיילר, או באגים במערכת ההפעלה אחת לחודש, אזי רוב המהנדסים לא יקדישו אמצעים להימנעות מקיפאון. ב- UNIX למשל, מספר התהליכים שבמערכת נקבע על ידי מספר הכניסות שבטבלת התהליכים.

לפיכך החריצים בטבלת התהליכים היא משאב סופי. אם FORK נכשל כי הטבלה מלאה, גישה הגיונית עבור תוכנית היא לבצע FORK ואז להמתין פרק זמן, ולנסות שוב.

עמוד 246

נניח כעת שבמערכת UNIX יש 100 חריצי תהליכים. 10 תהליכים רצים במקביל, וכל אחד צריך ליצור 12 תת-תהליכים. אחרי שכל תהליך יצר 9 תהליכים, עשרת התהליכים האורגינלים, ו- 90 החדשים, ימלאו את הטבלה. כל אחד מ- 10 התהליכים המקוריים ממתין עתה בלולאה אינסופית, מבצע FORK ונכשל – קיפאון. ההסתברות להתרחשות שכזו היא קטנה, אולם קיימת. בהתאמה, מספר הקבצים הפתוחים תלוי בטבלת I-NODE, ובעיה זהה יכולה להתרחש כאשר היא מתמלאת. מקום על הדיסק הוא משאב מוגבל נוסף. למעשה, כל טבלה במערכת ההפעלה מייצגת משאב מוגבל.

הגישה ב-UNIX היא להתעלם מהבעיה מתוך הנחה שרוב המשתמשים יעדיפו קיפאון מקרי, מאשר הגבלת המשתמשים לתהליך אחד, קובץ פתוח אחד וכו'. אם ניתן למנוע קיפאון בחינם, לא היה טעם לדיון. הבעיה היא שמחיר הפתרון הוא גבוה, במונחים של אי נוחות והגבלת תהליכים.

6.4 גילוי קיפאון והיחלצות ממנו

המערכת לא מנסה למנוע מראש מצב של קיפאון, אלא נותנת למצבים להתרחש, מנסה לאתר אותם כשהם מתרחשים ואז לטפל בהם. בפרק זה נדון במספר דרכים לאתר ולטפל במצב קיפאון. דרך אחת לאיתור כבר ראינו: על ידי בניית גרף הקצאת המשאבים.

6.4.1 גילוי קיפאון כאשר יש רק משאב יחיד מכל סוג

נתחיל במקרה הפשוט: רק משאב אחד מכל סוג קיים למשל למערכת יש מדפסת אחת, כונן טייפ אחד וכו'. למערכת כזו אנו בונים גרף משאבים, איור 6-1. אם הגרף מכיל מעגל אחד או יותר, יש קיפאון. כל תהליך שהוא חלק ממעגל, נמצא בקיפאון. אם אין מעגלים, המערכת לא בקיפאון.

<u>עמוד 247</u>

כדוגמא למערכת מסובכת יותר, נניח מערכת עם 7 תהליכים A עד G, וכן 6 משאבים R עד W. חלוקת המשאבים:

1. תהליך A מחזיק במשאב R ורוצה את S.
2. תהליך B לא מחזיק משאב, אולם רוצה את T.
3. תהליך C לא מחזיק משאב, אולם רוצה את S.
4. תהליך D מחזיק את U ורוצה את S ואת T.
5. תהליך E מחזיק את T ורוצה את V.
6. תהליך F מחזיק את W ורוצה את S.
7. תהליך G מחזיק את V ורוצה את U.

השאלה היא האם המערכת בקיפאון, ואם כן, איזה תהליכים מעורבים?
ניתן לענות על כך על סמך גרף המשאבים שבאיור 3-6. הגרף מכיל מעגל אחד שממנו אנו למדים שתהליכים D, G , E נמצאים בקיפאון. תהליכים A,C,F לא נמצאים בקיפאון כי S ניתן להקצאה לכל אחד מהם, שמסים ומחזיר אותו. אזי שניים אחרים יכולים לקחת אותו, לסירוגין.
אמנם קל יחסית לאתר תהליכים בקיפאון בגרף פשוט, אולם במערכת אמיתית יש צורך באלגוריתם פורמלי לאיתור קיפאון. האלגוריתם הבא מסתיים כאשר הוא מאתר מעגל, או אם לא קיים מעגל. האלגוריתם משתמש במבנה נתונים L שהוא רשימת צמתים. במהלך האלגוריתם, קשת תסומן להצביע על כך שנסרקה.

<u>עמוד 248</u>

1. עבור כל צומת בגרף (משאב או תהליך) יש לבצע את 5 הצעדים הבאים כאשר הצומת הנבחרת בכל שלב מהווה התחלה.
2. אתחל את L (רשימה מקושרת של הצמתים שעברנו בהם בגרף), ואתחל את כל החצים בגרף כלא מסומנים.
3. הוסף את הצומת הנוכחית לרשימה (לסופה) ובדוק האם היא כבר קיימת בה, אם כן הגרף כולל מעגל והאלגוריתם מסתיים.
4. מהצומת הנוכחית בדוק האם יש חצים שלא מסומנים. אם כן עבור לשלב 5, אחרת עבור לשלב 6.
5. בחר חץ לא מסומן באופן אקראי (אם יש כמה) וחזור על הצעדים עבור הצומת שאליו החץ מוביל מצעד 3.
6. כעת הגעת למבוי סתום. חזור אחורה לצומת האחרונה שטיפלת בה והפוך אותה לצומת הנוכחית – חזור לצעד 3. אם חזרת לצומת הראשונה – המקורית אזי הגרף לא כולל מעגל והאלגוריתם מסתיים.

שיטת חיפוש המעגלים בגרף הקצאת המשאבים היא חיפוש DFS מקובל.
האלגוריתם לוקח כל צומת, בתורה, כשורש של מה שהוא מקווה שיהיה עץ, ומבצע חיפוש לעומק עליו. אם הוא חוזר אחורה לצומת שאותה סרק, אזי נמצא מעגל. אם כל הקשתות נסרקו מכל צומת נתון, הוא חוזר על עקבותיו לצומת קודם. אם הוא חוזר אחורה עד לשורש ולא יכול להמשיך, אזי תת הגרף שנסרק מהצומת הנוכחית לא מכיל מעגלים. אם תכונה זו נשמרת עבור כל הצמתים, כל הגרף נקי ממעגלים והמערכת לא בקיפאון.
נראה כיצד האלגוריתם פועל, לפי הגרף באיור 3-6. סדר סריקת הצמתים הוא אקראי, אזי נלך משמאל לימין, מלמעלה למטה. נתחיל ב- R ואחר כך A,B,C,S,D,T,E,F וכן הלאה. אם מגיעים למעגל האלגוריתם עוצר.
L מאותחל לרשימה ריקה. נוסיף את R לרשימה, ונתקדם לאפשרות היחידה A ונוסיף אותה ל-L. מ-A נמשיך ל-S. משם אין קשתות יוצאות אזי נחזור ל-A. היות וכל הקשתות של A מסומנות, נחזור ל- R ונסיים.
עתה נתחיל את האלגוריתם ב-A, נאתחל את L לרשימה ריקה. סדר הסריקה הוא B,T,E,V,G,U,D. כעת נבצע בחירה מקרית ונבחר את S, נגיע למבוי סתום ונחזור ל- D. בפעם השניה נבחר T ונוסיף לרשימה. אם נגלה מעגל ייעצר האלגוריתם.

עמוד 249\250

6.4.2 גילוי קיפאון כאשר המשאבים מחולקים למחלקות

א. סימונים.

1. ישנם N תהליכים, והם מסומנים במספרים 1 עד N.
2. ישנן M מחלקות משאבים המסומנות במספרים 1 עד M.

ב. מבנה הנתונים.

1. P הוא מערך בוליאני המסמל את מצב התהליכים. כל תהליך Pi יכול להיות במצב שבו כל בקשותיו מסופקות (1) או במצב של המתנה למשאבים (0). בתחילת האלגוריתם, ערך כל התאים במערך הוא 0
2. E הוא מערך חד-ממדי בגודל M. בכל תא Ej מאוחסן מספר המשאבים הכללי במערכת השייכים למחלקה j. כל המשאבים הנמצאים באותה מחלקה שקולים זה לזה מבחינת התהליכים הדורשים אותם.
3. A הוא מערך חד-ממדי בגודל M. בכל תא Aj מאוחסן מספר המשאבים החופשיים שלא הוקצו עדיים מהמחלקה j.
4. C הוא מערך דו-ממדי בגודל N*M. בתא Cij מאוחסן מספר המשאבים מהמחלקה j שכבר הוקצו לתהליך i.
5. R הוא מערך דו-ממדי בגודל M*N. בתא Rij מאוחסן מספר המשאבים מהמחלקה j הנדרשים עדיין על ידי התהליך i.

ג. מכיוון שכל משאב יכול להיות או חופשי או מוקצה, הרי שהסכום של מספר המשאבים מהמחלקה j שכבר הוקצו לאחד התהליכים ומספר המשאבים ממחלקה זו שעדיים חופשיים הוא תמיד Ej.

ד. אנו מגדירים יחס => בין שני וקטורים A B שווים באורכם על ידי : A<=B כאשר לכל I מתקיים Ai<=Bi.

ה. שלבי האלגוריתם

1. מחפשים תהליך I שנמצא במצב של המתנה למשאבים (PI=0) ואשר הווקטור Ri המייצג את בקשותיו (וקטור השורה I במערך R) קטן או שווה לווקטור A המייצג את המשאבים הזמינים (Ri<=A). במילים אחרות אנו מחפשים תהליך הממתין למשאבים, שניתן להיענות לכל בקשותיו. תהליך כזה יכול להמשיך ולסיים את פעולתו ללא חשש מקיפאון.
2. אם מצאנו תהליך Pi כזה, אנו מחברים חיבור וקטורי ומציבים : A=A+Ci. כמו כן אנו משנים את ערכו של Pi ל- 1. במילים אחרות, התהליך ממשיך ומסתיים ואז מחזיר את כל המשאבים שהחזיק בהם לשימושם של תהליכים אחרים.
3. כאשר אין יותר תהליך I כנדרש בשלב 1, האלגוריתם מסתיים.

אם לאחר סיום האלגוריתם ישנם תהליכים שבקשותיהם לא נענו, תהליכים אלו נמצאים במצב של קיפאון.
כמו כן ייתכן שהאלגוריתם יסתיים ללא ביצוע שלב 2, כאשר כל התהליכים נמצאים במעגל הקיפאון לא יתבצע שלב 2 כי אין תהליך Pi כנדרש בשלב 1.

עמוד 251

השאלה היא מתי יש לחפש את מצבי הקיפאון?
דרך אחת היא לחפש בכל פעם שמוגשת בקשה למשאב. בדרך זו נגלה מצבי קיפאון מוקדם אך דרך זו יקרה במונחי CPU. דרך חלופית היא לבדוק כל K דקות או שמספר תהליכים מסוים כבר נכנס למצב קיפאון.

6.4.3 היחלצות מקיפאון

נתאר בסעיף זה שלוש דרכים להיחלצות מקיפאון. שים לב למחיר הכבד של כל דרך.

<u>החלמה על ידי חילוץ כפוי</u>
קיפאון נוצר כתוצאה מהקצאת משאבים, ולכן חילוץ מבוקר של משאבים עשוי לחלץ גם מקיפאון. החילוץ הוא כפוי שכן הוא נעשה בהוראת מערכת ההפעלה ולא ביוזמתו של התהליך.
מדוע אין אפשרות לחכות עד שהתהליך ישחרר את המשאב עם סיום הצורך בו? מעצם הגדרתו של מצב הקיפאון נובע כי התהליכים לעולם לא יגיעו לשלב של שחרור משאב כלשהו.
במקרים מסוימים ניתן לקחת באופן זמני משאב מהבעלים שלו, ולתת אותו לתהליך אחר. לדוגמא, כדי לקחת מדפסת לייזר מהבעלים שלו, המפעיל יכול לאסוף את הדפים שהודפסו כבר, ולשים אותם בערמה. אזי ניתן להשעות את התהליך. בשלב זה ניתן להקצות את המדפסת לתהליך אחר. כאשר תהליך זה מסיים, ניתן להחזיר את הערימה של הדפים המודפסים למדפסת, והתהליך המקורי יכול להמשיך.
היכולת לקחת משאב מתהליך, ולתת אותו לאחר, ואז להחזירו מבלי שהתהליך ישים לב, תלוי במידה רבה במשאב. החלמה בדרך זו בדרך כלל קשה או בלתי אפשרית. בחירת התהליך שיש להשעות תלויה במידה רבה בתהליך שמחזיק את המשאב שהכי קל לקחת בחזרה.
חילוץ כפוי הוא התערבות גסה, העלולה לגרום נזק חמור למהלך התקין של התהליך שאולץ לוותר על משאב. לכן לעתים קרובות נדרשת התערבות אנושית כדי לצמצם את ממדי הנזק.
דוגמאות לנזק שעלול להיגרם על ידי חילוץ כפוי:

א. השארת מסד נתונים במצב בלתי עקבי.
ב. ניתוק קו תקשורת במהלך קבלת הודעות.
ג. איבוד תוצאות של חישוב ארוך בגלל אי היכולת לשמור אותן בקובץ.

<u>עמוד 252</u>

<u>היחלצות על ידי גלגול לאחור – ROLLBACK</u>
שיטה זו אינה ייחודית להיחלצות מקיפאון. הבעיה של שחזור המצב אחרי תקלה פתאומית היא בעיה מוכרת וידועה. פתרון מקובל הוא שמירה תמידית של מצבים בטוחים (CHECKPOINTS). כאשר המערכת קורסת, התהליכים מתחילים מחדש ממצב ידוע ואינם חייבים לחזור לנקודת ההתחלה.
התהליכים לא יכולים להמשיך מהמקום בו הופסקו היות וללא נקודות ציון מוסכמות מראש, אין שום אפשרות לשחזר את מצבו של תהליך. ממש כשם שקובץ שעריכתו נקטעה על ידי הפסקת חשמל ניתן לשחזור רק על פי הגרסה האחרונה שנשמרה על הדיסק. מנגנון הגלגול לאחור מאפשר לצמצם את הנזק הנגרם על ידי חילוץ כפוי.
במערכות שידוע שמתרחשים מצבי קיפאון ניתן לארגן תהליכים מיוחדים הקרויים תהליכי מצבים בטוחים – CHECKPOINTS. תהליכים אלה משמעותם שמצב התהליך (STATE) נרשם בקובץ, כך שהוא יכול להיות משוחזר לאחר מכן. השמירה מתבצעת לא רק למצב הזיכרון אלא גם למצב המשאבים שקשורים לתהליך, כלומר איזה משאב מוקצה כעת לתהליך. כדי שזה יהיה אפקטיבי, תהליכים המצבים הבטוחים צריכים לרשום את המידע כל פעם בקבצים חדשים ולא לדרוס ישנים, כך שכאשר תהליך מתבצע צוברים את כל רצף קובצי המצבים הבטוחים.
כאשר נוצר מצב קיפאון, קל לראות לאיזה משאב זקוקים. כדי להתאושש, תהליך שמחזיק במשאב הדרוש מגולגל לאחור עד לנקודת זמן שלפני בקשת כמה משאבים אחרים על ידי הפעלתו החל מאחד מהמצבים הבטוחים. כל העבודה שהתבצעה אחרי המצב הבטוח אובדת. למעשה התהליך מאותחל למצב מוקדם יותר שבו לא החזיק במשאב, שכעת מוקצה לאחד מהתהליכים שהיו בקיפאון. אם התהליך יבקש את המשאב שוב, הוא יאלץ לחכות עד אשר המשאב יתפנה.
<u>היחלצות על ידי הריגת תהליכים</u>
זוהי גרסה נוספת של חילוץ כפוי. מותו של התהליך איננו מעניין אותנו כשלעצמו, אבל הידיעה כי עם מותו ישתחררו המשאבים שהוא מחזיק בהם, מעודדת את ההריגה.
הקורבן אינו חייב להיות דווקא תהליך קפוא. העיקר הוא שכך ישתחרר משאב החיוני לשבירת הקיפאון. לגישה זו יש יתרון מכיוון שמבין התהליכים המיועדים לחיסול ניתן לבחור את זה שהריגתו תסב רק את הנזק המינימלי.

האם כל הריגת תהליך גורמת נזק? כן. לפחות מבחינת הזמן שבוזבז.
ניתן להתחיל בהריגת תהליך אחד במעגל, מתוך תקווה שאחרים יוכלו להמשיך. אם זה לא עוזר ניתן לחזור על התהליך עד שפותרים את המצב.
האלטרנטיבה היא להרוג תהליך שלא נמצא במעגל, כדי לשחרר את המשאבים שהוא מחזיק בהם. לפי גישה זו, התהליך נבחר כי הוא מחזיק במשאבים שתהליכים שנמצאים במעגל הקיפאון זקוקים להם.
עדיף להרוג תהליך שניתן להחזירו מהתחלה ללא נזק. לדוגמא, קומפילציה תמיד ניתן להחזיר כי הוא רק קורא קובץ מקור, ומיצר קובץ פלט. אם נהרוג אותו, הריצה הראשונה (שהופסקה) לא תשפיע על הריצה השניה.
מאידך, תהליך שמעדכן מאגר נתונים לא יהיה ניתן להריצו בשנית באופן בטוח. אם התהליך הוסיף 1 לרשומה כלשהי, בריצה הראשונה, לאחר הריגתו, כאשר נריץ אותו בשנית, הוא יוסיף 2 לאותה רשומה והדבר שגוי.

6.5 התחמקות מקיפאון

הואיל ומחירה של היחלצות מקיפאון הוא כבד, ייתכן שראוי לנסות ולהקצות משאבים בדרך שתבטיח מראש כי המערכת לא תיכנס לקיפאון. מערכת ההפעלה היא זאת שמקצה את רוב המשאבים במחשב, לכן יש מקום להניח כי מדיניות נבונה יכולה לשמור על התהליכים מפני כניסה לקיפאון. אם נמצא מדיניות הקצאה בטוחה מסוג זה, נוכל ליישם אותה בכל מקרה שבו קיימת סכנת קיפאון, גם כאשר המשאבים אינם מוקצים על ידי מערכת ההפעלה.
משאבים שאינם מוקצים על ידי מערכת ההפעלה הם בעיקר משאבים לוגיים כמו משתנים משותפים, במערכות המאפשרות זאת.
במקרה האחרון מספיק שכל קבוצה של תהליכים המשתפים ביניהם משאבים לוגיים תמנה לעצמה מנהל, שיקצה את המשאבים בהתאם למדיניות שנקבעה. זאת כמובן בתנאי שאכן קיימת מדיניות הקצאה המבטיחה התחמקות מקיפאון.

עד הנחנו שכאשר תהליך מבקש משאבים, הוא מבקש את כולם באותו זמן (שרטוט 4-6). כלומר עבור כל תהליך יודעים כמה משאבים היא מחזיק ואם ירצה אותם בעתיד. ברוב המערכות המצב אינו כזה, והתהליך מבקש משאב בכל פעם.
על המערכת להחליט האם הבקשה למשאב מכניסה את המערכת למצב לא בטוח, ולבחור במדיניות היכולה לשמור על התהליכים מפני כניסה לקיפאון.
נשאלת השאלה האם יש אלגוריתם שיכול תמיד למנוע קיפאון, על ידי בחירה נכונה?
התשובה היא כן – רק אם מידע מסוים יהיה ידוע מראש.
בפרק זה נבדוק דרכים להימנעות מקיפאון על ידי הקצאה נכונה של המשאבים.

עמוד 253

6.5.1 מסלולים להקצאה בטוחה של משאבים – RESOURCE TRAJECTORIES

מסלול בטוח: כל הצירופים של צעדי הקצאות ושחרור משאבים עבור קבוצת תהליכים, יוצרים מרחב מצבים. מסלול בטוח הוא מסלול במרחב זה העובר ממצב ההתחלה של התהליכים אל המצב שבו כולם מסיימים מבלי לחצות מצב שבו ייתכן קיפאון.
מסלול בטוח מאפשר סדרת הקצאות ושחרורים המאפשרת השלמת התהליכים בלי קיפאון.
שרטוט 6-6 מציג דיאגרמה של שני תהליכים ושני משאבים. הציר האופקי מיצג את מספר ההוראות שבוצעו על ידי תהליך A. הציר האנכי מייצג את מספר ההוראות שבוצעו על ידי תהליך B. ב- I1 A ביקש משאב א. ב- I2 הוא נזקק למשאב ב. שני המשאבים שוחררו ב- I3 I4. תהליך B זקוק למשאב ב מ- I5 עד I7 ואת המשאב א מ- I6עד I8. כל נקודה בדיאגרמה מייצגת את שני התהליכים. בתחילה המצב הוא P, שבו אף תהליך לא ביצע כל הוראה. אם המתזמן יריץ את A תחילה נגיע לנקודה Q, שבה A מבצע מספר הוראות, אולם B לא מבצע דבר. בנקודה q המסלול הופך להיות אנכי, מה שמצביע שהמתזמן מריץ את B.

עמוד 254

עם מעבד אחד כל המסלולים חייבים להיות או אנכיים או אופקיים, אך לא באלכסון. מעבר לכך, התנועה היא צפונה או מזרחה, לא דרום או מערב (תהליך לא יכול לנוע אחורה).

כאשר תהליך A חוצה את שורה I1 במסלול מ- r אל s הוא מבקש ומקבל את משאב א.

כאשר B מגיע לנקודה t, הוא מבקש את משאב ב. התחום המוצלל מעניין במיוחד. הוא מיצג את שני התהליכים מחזיקים במשאב א. דבר שאינו אפשרי.

אם המערכת תכנס לתחום המוגדר על ידי I1 I2 מהצדדים ו- I5 I6 מלמעלה ולמטה, נגיע לקיפאון כאשר נגיע להוראות I2 I6. בנקודה זו A מבקש את משאב ב ו- B מבקש את משאב א, ושניהם מוקצים כבר. כל האזור לא בטוח. בנקודה t הדבר הבטוח היחיד הוא להריץ את תהליך A עד שיגיע ל- I4. יותר מכך, כל מסלול ל- u יספיק.

הדבר החשוב הוא שבנקודה t B מבקש משאב. המערכת צריכה להחליט האם לאשר אותו, או לא. אם כן המערכת תיכנס לאזור לא בטוח שיסתיים בקיפאון. כדי להימנע מכך B יושעה עד ש- A ישחרר את משאב ב.

6.5.2 מצבים בטוחים לעומת מצבים מסוכנים

מצב נקרא בטוח אם התהליכים בו אינם קפואים ויש דרך להשלים את כל הבקשות של התהליכים כך שכולם יסתיימו בוודאות ללא קיפאון.

מצב מסוכן אם ייתכן שהתהליכים יכנסו לקיפאון. כלומר ייתכן שהם לא יכנסו לקיפאון, אך אין ערובה לכך.

שרטוט 7-6 מדגים מצב בטוח מכיוון שעל ידי תזמון תהליכים טוב נמנע מצב של קיפאון. כל התהליכים סיימו ללא קיפאון וכל הבקשות שלהם מולאו.

בעמוד 255 שרטוט 8-6 מדגים מצב שאינו בטוח, זאת מכיוון שעל ידי המעבר ממצב a למצבי b ומילוי הבקשה של A המערכת הגיעה למצב של קיפאון לתהליך A,C. לכן למעשה על ידי תזמון שגוי – מילוי בקשת A המערכת נכנסה למצב מסוכן. ייתכן שעל פי אותם מצבים המערכת לא תגיע למצב של קיפאון. אך עצם זה שלא בוודאות – המצב מסוכן.

עבור אלגוריתם זה יש צורך לדעת מראש מה מקסימום המשאבים שכל תהליך ידרוש.

עמוד 255

6.5.3 אלגוריתם הבנקאים עבור משאב יחיד

בבנק יש סכום כסף נתון, אך מכיוון שלא כל לקוחותיו הלווים זקוקים לכל סכום הכסף הזה בעת ובעונה אחת, הוא יכול להקצות ללווים קווי אשראי הגדולים מסך כל הכסף המופקד בו (ובכך להגדיל את רווחיו). ברור כי במצב זה הבנק חשוף לסכנה של פשיטת רגל (למשל, אם כולם ינצלו את מלוא האשראי בו-זמנית). כדי להיות במצב בטוח, על הבנק להחזיק תמיד סכום כסף נזיל שיאפשר לו להיענות לצרכים הצפויים של הלווה הבא. על העיקרון הזה בנוי אלגוריתם הבנקאים.

חוקי המשחק הם:

1. לכל לווה (תהליך) יש קו אשראי אישי מוגבל (מספר משאבים מקסימלי שהוא רשאי לדרוש).
2. כאשר לווה ממצה את כל בקשותיו לאשראי (משאבים), הלווה פורע את כל הלוואותיו לבנק (התהליך מסתיים ומחזיר את משאביו למערכת ההפעלה).
3. לבנק מותר לא להיענות לבקשות אשראי באופן זמני, אך הוא חייב להיות תמיד במצב המאפשר לפחות לאחד הלווים למצות את בקשותיו (וכך יוכל להחזיר את כל הכסף לבנק).

ברור כי אם חוק 3 יישמר, הבנק לעולם לא יפשוט רגל (והתהליכים לעולם לא יקפאו). חוק זה הוא תמציתו של אלגוריתם הבנקאים.

כדי לבדוק אם מצב הוא בטוח, מנהל המשאבים בודק אם יש לו מספיק משאבים לסיפוק צרכיו של תהליך כלשהו. אם התשובה שלילית, המערכת כבר נמצאת בקיפאון. אם התשובה חיובית, התהליך המבקש מקבל את ההקצאה הדרושה לו, והבדיקה נעשית לגבי

תהליך חדש. אם בדרך זו ניתן לסיים בצורה תקינה את פעולת כל התהליכים, המצב נחשב לבטוח.
במילים אחרות, מצב נחשב לבטוח אם קיימת אפשרות אחת לפחות של הרצה סדרתית של התהליכים (הרצת התהליכים בזה אחר זה אך לא במקביל) כך שכולם יסתיימו.
ההרצה הסדרתית היא המוצא האחרון מפני קיפאון ולכן, כל עוד אפשרות זו קיימת, המצב נחשב לבטוח. שים לב כי אין פירוש הדבר שמעתה והלאה ההרצה תהיה בהכרח סדרתית בלבד, שכן הדבר תלוי בסדר הדרישות של התהליכים, אולם מצב בטוח מבטיח שגם במקרה של דרישות מקסימליות, לא ייווצר קיפאון – ולו על ידי הרצה סדרתית.
אם בשלב כלשהו של האלגוריתם המערכת נכנסת לקיפאון, המצב אינו בטוח. ופרושה של הטענה האחרונה הוא כי סדר בחירת התהליכים מבין אלו שניתן להריצם, אינו משנה דבר!

סיבוכיות האלגוריתם: כאשר יש N תהליכים, האלגוריתם לבדיקת המצב הבטוח מסתיים לאחר N צעדים, כאשר כל צעד הוא הקצאה ושחרור של משאבי תהליך יחיד.

עמוד 256

6.5.4 אלגוריתם הבנקאים למחלקות של משאבים

האלגוריתם במקרה זה דומה מאוד לאלגוריתם בסעיף הקודם. רק מבני הנתונים הם שונים. כמו כן אנו נדרשים לקבוע סדר שרירותי של סוגי המשאבים הקיימים. בדוגמה שבאיור 6-10 הסדר הוא:

1. TAPES
2. PLOTTERS
3. PRINTERS
4. CD-ROM DRIVERS

עתה דרושים לנו מבני הנתונים הבאים:

1. מערך דו ממדי C המכיל לכל תהליך את רשימת אחזקותיו עבור כל סוג של משאבים.
2. מערך דו-ממדי R המכיל לכל תהליך את רשימת צרכיו לגבי כל סוג משאבים.
3. וקטור E המכיל את המלאי הקיים במערכת מכל משאב.
4. וקטור P המכיל את הכמות התפוסה מכל משאב.
5. וקטור A המכיל את הכמות הפנויה מכל משאב.

על פי הגדרה ברור כי תמיד מתקיים A+P=E.
מכאן ואילך האלגוריתם הוא פשוט. שים לב כי גם כאן אין חשיבות לסדר התהליכים הנבחרים בשלב הראשון, ובלבד שמשאבי המערכת יספיקו כדי שתהליכים אלה יוכלו להסתיים.
אלגוריתם הבנקאים מתאים למערכת סגורה, שבה כל התהליכים, צורכיהם וכמות המשאבים ידועים מראש. מערכת ההפעלה אינה מערכת סגורה מבחינה זו. מספר התהליכים משתנה בהתמדה. בדרך כלל, הצרכים של תהליך אינם ידועים מראש, וגם כמות המשאבים משתנה בהתמדה. לפיכך, אלגוריתם הבנקאים הוא פתרון תיאורטי נאה, אך לא ניתן ליישם במקרה הכללי של מערכות הפעלה.

עמוד 257

בדוגמה שבספר יש 3 TAPES, 3 PLLOTERS, 4 PRINTERS, 2 CD-ROM.
האלגוריתם לבדיקת קיום מצב בטוח (לפי שרטוט 6-10):

1. הסתכל בשורה R, שסך המשאבים הדרושים לתהליך קטן\שווה ל- A. אם שורה כזו לא קיימת, המערכת תסתיים בקיפאון, כי אף תהליך לא יכול לרוץ עד קומפילציה.
2. נניח שהתהליך שבשורה שנבחרה מבקש את כל המשאבים להם הוא זקוק (ומובטח שניתן להקצות לו אותם) ומסיים. סמן את התהליך כאשר הוא מסיים, והוסף את כל המשאבים שהחזיר לוקטור A.
3. חזור על שני השלבים הראשונים עד שכל התהליכים מסומנים, בכל אחד מהמקרים הללו המצב ההתחלתי היה בטוח, או עד שמתרחש קיפאון, במקרים שהמצב לא היה בטוח.

אם ישנם כמה תהליכים שניתן לבחור בהם בשלב 1, לא משנה במי לבחור: סך המשאבים הפנויים או שתגדל, או שתישאר זהה.
בדוגמא, המצב הנוכחי בטוח. נניח שתהליך B מבקש כעת מדפסת. ניתן למלא אחר בקשתו כי לאחר מילוי הבקשה המצב יישאר בטוח. (תהליך D יכול לסיים, ואז תהליך A או E, והשאר אחריהם).
נניח עתה שאחרי שנתנו ל- B אחד מתוך שתי המדפסות הנותרות, E מבקש את המדפסת האחרונה. אישור הבקשה יקטין את וקטור המשאבים הפנויים
ל- (0 0 0 1), מה שיוביל לקיפאון. ברור שיש להשעות את בקשת E באופן זמני.

עמוד 258

6.6 מניעת קיפאון

התחמקות מקיפאון היא עניין מסובך מדי לצרכים מעשיים, שכן היא דורשת את ידיעת העתיד. מסתבר כי אילו הייתה לנו יכולת לחזות את העתיד, לא רק בעיות הקיפאון היו נפתרות אלא גם בעיות רבות אחרות. מכל מקום כבר נתקלנו לא פעם באפשרות ליצור אלגוריתם אופטימלי בתנאי שהעתיד ידוע.
למשל אלגוריתם תזמון אופטימלי, ואלגוריתם החלפת דפים אופטימלי.
בתחילת הדיון הזכרנו ארבעה תנאים הכרחיים לקיומו של קיפאון. הניסיונות הבאים למניעת קיפאון מבוססים כולם על תקיפת תנאים אלו.

6.6.1 תקיפת תנאי המניעה ההדדית

ברור כי אם כל משאב ניתן לחלוקה אזי לא ייתכן קיפאון. מצד שני ברור גם כי לא כל משאב ניתן לחלוקה, והדוגמה המובהקת לכך היא כאמור המדפסת, כך מסתבר כי בעזרת מנגנון מתאים (כאן השד המדפיס) ניתן לעתים להפוך משאב בלעדי למשאב הניתן לחלוקה. לפי מודל זה התהליך היחיד שמבקש את המדפסת הוא השד המדפיס, שאר התהליכים יכולים לשלוח הדפסות לאותה מדפסת בזמן נתון, והשד הוא היחיד שמפנה אותן למדפסת לפי הסדר. היות והשד לא מבקש כל משאב אחר, אנו יכולים להימנע מקיפאון עבור המדפסת.
לדאבוננו, לא כל ההתקנים יכולים להשתמש במנגנון זה – למשל קטע קריטי.
כמו כן התחרות על שטח הדיסק יכולה בעצמה להוביל למצב של קיפאון כאשר תהליך יתפוס את חצי שטח ה-SPOOLING ותהליך שני את החצי השני, ואף אחד מהם לא מסיים. אם השד מתוכנת להתחיל להדפיס לפני שהמידע עבור ל- SPOOLER, אזי המדפסת עלולה להישאר ללא תעסוקה אם תהליך יחליט להמתין כמה שעות לאחר החלק הראשון של ההדפסה. אף שני התהליכים לא יסיימו וכך נגיע לקיפאון. מסיבה זו, שדים מתוכנתים להדפיס רק לאחר שכל קובץ הפלט פנוי.
הרעיון הוא להקצות משאב רק אם נחיצותו אבסולוטית, ולנסות להבטיח שמעט תהליכים ככל האפשר יבקשו משאב.
האם דיסק הוא משאב בלעדי? כיצד הוא מתחלק בין תהליכים שונים?
למרבה הצער, לא תמיד קיימת אפשרות כזאת. דוגמה טובה לכך הוא הקטע הקריטי שהוא משאב בלעדי על פי עצם הגדרתו. כמו כן הפתרון שהובא בהקשר של המדפסת מתבסס על העובדה כי ניתן לדחות את ההדפסה, ובינתיים להמשיך בפעולת התהליך. פתרון זה הוא כמובן בלתי אפשרי כאשר המשך התהליך תלוי בתפיסה מיידית של המשאב. מהדיון עולה היוריסטיקה (כלל שכדאי לנהוג על פיו אך תועלתו איננה מוכחת בכל תנאי) והיא:

א. לנסות להפוך משאבים בלעדיים הניתנים לדחייה, למשאבים הניתנים לחלוקה.
ב. לא לדרוש משאב אלא כן הוא הכרחי לחלוטין.

חולשתה הגדולה של היוריסטיקה הזו נעוצה בעובדה שהיא תלויה ברצונם הטוב של התהליכים (או ליתר דיוק במשתמשים שמריצים אותם). למרבה הצער, בעולמנו קשה עדיין לבטוח בפתרון המבוסס על רצון טוב בלבד.
ניתן בכל זאת להשתמש בהיוריסטיקה הזו כאשר מנסים למנוע קיפאון בתוך קבוצת תהליכים שכולם תוכננו מראש לשתף פעולה. למשל, כאשר קבוצת תהליכים מפעילה מסד נתונים. במקרה כזה, כאשר המטרה היא משותפת ואין ניגודי אינטרסים, יש אפשרות מעשית להקפיד על כללים נאותים שיצמצמו עד מאוד את סכנת הקיפאון.

מאחר שהיוריסטיקה זו תלויה בשיתוף פעולה מרצון, הדבר מקטין עד מאוד את יעילות השימוש בה.

עמוד 259

6.6.2 תקיפת תנאי ההחזקה וההמתנה

התנאי השני הוא המתנה. אם נוכל למנוע מתהליך המחזיק במשאבים להמתין למשאבים אחרים, נמנע מצב של קיפאון.
דרך אחת להשיג מטרה זו היא לדרוש מכל התהליכים לבקש את כל המשאבים הדרושים להם לפני תחילת הביצוע. אם המשאבים פנויים הם יוקצו לתהליך, והוא יוכל לרוץ עד לקומפילציה. אחרת, אם משאב אחד או יותר, לא פנויים, לא יוקצה דבר לתהליך והוא ימתין עד שיתפנו.
הבעיה המיידית של גישה זו היא שרוב התהליכים אינם יודעים מראש איזה משאבים יצטרכו על לתחילת הריצה. למעשה, אם ידעו מראש, יכולנו להשתמש באלגוריתם הבנקאים.
בעיה נוספת היא שהמשאבים לא יהיו מנוצלים בצורה אופטימלית. לדוגמא, תהליך שקורא נתונים מתוך טייפ, מנתח אותם במשך שעה, ואז מדפיס, התהליך יתפוס את המדפסת שעה לחינם.
הקצאה מראש של כל המשאבים הבלעדיים יוצרת בזבוז משאבים עצום שכן לעיתים התהליכים נאלצים להחזיק במשאבים זמן רב בטרם יזדקקו להם בפועל. מאחר שמשאבים אלה בלעדיים, במשך כך הזמן הזה, שום תהליך אחר לא יוכל להשתמש בהם (יש לכך השפעה על דרגת המקביליות במערכת ההפעלה).
למרות זאת ישנם מערכות אצווה הדורשות מהמשתמש רשימה של כל המשאבים, בשורה הראשונה כל של תוכנית. המערכת משיגה את המשאבים, ושומרת אותם עד שהתוכנית מסתיימת. שיטה זו מקשה על המשתמש, ומבזבזת משאבים, אולם מונעת קיפאון.
דרך נוספת היא לשבור את תנאי HOLD-AND-WAIT היא לדרוש מתהליך שמבקש משאב, לשחרר תחילה את כל המשאבים שהוא מחזיק, ואז לנסות לבקש את כל המשאבים להם הוא זקוק בבת אחת.

6.6.3 חילוץ כפוי

בחסרונות שיטה זו עסקנו בסעיף 6.4.3. מחירה גדול מדי לכל יישום כללי.

6.6.4 מניעת המתנה מעגלית

ניתן לתקוף תנאי זה במספר דרכים:

א. הקצאת משאב יחיד – על ידי חוק האומר שתהליך יכול להחזיק רק במשאב אחד. אם הוא זקוק למשאב שני, עליו לשחרר את הראשון. עבור תהליך שצריך להעתיק קובץ ענק מטייפ למדפסת, חוק זה אינו מעשי. דרך זו למרות שהיא בטוחה מבחינה תיאורטית, היא אינה מעשית.

ב. הקצאת משאבים בסדר קבוע מראש - בשיטה זו מסדרים את המשאבים בסדר מסוים וממספרים אותם. עתה תהליך יכול לבקש את כל המשאבים להם הוא זקוק, אולם הבקשות צריכות להיות לפי הסדר נומרי. אין להקצות משאב שמספרו נמוך אחרי משאב שמספרו גבוה יותר. בדרך זו גרף הקצאת המשאבים לא יוכל להיות מעגלי – (שרטוט 6-11) נניח שתהליך A ביקש משאב J, ותהליך B ביקש את משאב I. אם I>j אזי A לא יכול לבקש את j. אם I<j אזי B לא יכול לבקש את i. בכל מקרה נמנע קיפאון. כאשר יש מספר רב יותר של תהליכים, אותה לוגיקה נשמרת. בכל רגע, אחד מהמשאבים המוקצים יהיה הגבוה ביותר. התהליך המחזיק במשאב לא יבקש משאב שכבר תפוס. או שהוא יסיים, או שהוא יבקש משאב בעל מספר סידורי גבוה יותר. לבסוף הוא יסיים וישחרר את המשאב. בשלב זה תהליך אחר יחזיק עתה במשאב עם מספר סידורי הגבוה ביותר, ואף היא יוכל לסיים.

ג. וריאציה נוספת של אלגוריתם ב היא להוריד את הדרישה שהמשאבים יוקצו בסדר עולה של מספרים, ולדרוש שתהליך לא יוכל לדרוש משאבים שערכם נמוך יותר משל אלו שהוא מחזיק בהם. אם תהליך ביקש את משאב 9 ומשאב 10, ואז משחרר אותם, רק אז הוא יוכל לבקש את משאב 1.

למרות שסדר מספרי של משאבים מונע קיפאון קשה ליישמו מכיוון שיהיה קשה למצוא סדר תהליכים כזה שיספק את כולם. מספר המשאבים הפוטנציאלי והמשתמשים השונים יכול להיות כה עצום עד ששום סדר לא יוכל לעבוד.

עמוד 261

6.7 נושאים נוספים הקשורים לקיפאון

6.7.1 נעילה בשני מעברים – TWO-PHASW LOCKING

עד כה ראינו כי שיטות ההתחמקות מקיפאון והדרכים שהוצעו למניעתו הן בעלות מחיר כה כבד עד שיישומן הופך במקרים רבים לבלתי כדאי. אולם במקרים מיוחדים, בעיקר כאשר התהליכים המעורבים משתפים פעולה, ואין חשש כי ינסו לחטוף משאבים, ניתן להציע פרוטוקולים סבירים להימנעות מקיפאון. פרוטוקול הנעילה בשני מעברים הוא אולי הפרוטוקול הידוע ביותר מסוג זה, והוא אכן פותח לצרכים של מסדי נתונים. מצב אופייני במערכות מסדי נתונים הוא שמספר תהליכים מעדכנים רשומות במקביל. אולם רשומה נכתבת היא משאב בלעדי, ולכן המערכת כולה חשופה לסכנה של קיפאון.

אלגוריתם הנעילה בשני מעברים – במעבר הראשון התהליך מנסה לנעול את כל הרשומות שהוא צריך, כל אחת ואחת. אם הוא מצליח הוא מתחיל לעבור במעבר שני, מעדכן ומשחרר את הנעילה. כל עבודה ממשית לא מתבצעת במעבר הראשון.

אם במהלך המעבר הראשון, רשומה מסוימת שהוא זקוק לה כבר נעולה, התהליך משחרר את כל הרשומות שהוא נעל, וחוזר לשלב הראשון. גישה זו דומה לבקשת כל המשאבים מראש. בוריאציות מסוימות של אלגוריתם שני המעברים, כאשר תהליך נתקל ברשומה נעולה במעבר הראשון, הוא לא משחרר את כל הרשומות שהוא נעל. בוריאציה זו יכול להתרחש קיפאון.

בכל מקרה, שיטה זו אינה מקובלת באופן כללי. היות ובמערכות זמן אמת לדוגמא, אין זה מקובל שפשוט עוצרים תהליך באמצע הפעולה בגלל שמשאב מסוים אינו זמין, ומאלצים אותו להתחיל מהתחלה, וגם כל פעולה אחרת שלא ניתן להפסיק ולהתחיל מהתחלה באופן בטוח, כמו קריאה וכתיבת הודעה ברשת, עדכון קובץ וכדומה.

האלגוריתם שימושי רק כאשר למתכנת יש סדר ברור של צעדים, וניתן לשחזרם.

6.7.2 קיפאון שאינו קשור במשאבים מקובלים

גם אירוע הוא משאב לצורך קיומו של קיפאון. קיפאון מסוג זה יכול להיווצר כאשר שני תהליכים ממתינים כל אחד לסיום פעולתו של האחר, ולכן שניהם נמצאים בקיפאון. קיפאון מסוג זה אופייני לשגיאות תוכנה. בניגוד לקיפאון התלוי בהקצאת משאבים על ידי מערכת ההפעלה, כאן מערכת ההפעלה אינה מעורבת כלל, ולכן היא איננה יכולה לטפל בגורמים של קיפאון מסוג זה.

כיצד יכולה מערכת ההפעלה להתמודד עם קיפאון שאינה יכולה לאתרו?

אם לא ניתן לטפל בגורמי המחלה, כל שנשאר לנו הוא לטפל בסימפטום, והסימן הברור ביותר לקיומו של קיפאון הוא שהתהליכים אינם מתקדמים. כדי לבדוק את הסימפטום, מערכת ההפעלה יכולה לעקוב אחר התקדמות התהליכים, ולהזעיק את מנהל המערכת במקרה שמתגלים תהליכים אשר אינם מתקדמים כלל במשך תקופה ארוכה.

למשל בפרק 2 ראינו דוגמא שבה תהליך צריך לבצע DOWN בשני סמפורים, mutex טיפוסי ועוד אחד. אם הם יבוצעו בסדר שגוי, יתכן ויתרחש קיפאון.

6.7.3 הרעבה – STARVATION

הרעבה במקרה הכללי היא מצב שבו תהליך איננו יכול להמשיך במהלכו התקין בגלל המתנה ארוכה מאוד (או אינסופית) למשאב אחד לפחות. לפיכך קיפאון הוא מקרה פרטי של התופעה הכללית של הרעבה. קיפאון מתייחד גם בכך שלתהליך קפוא יש חלק במצב שהביא אותו לשיתוק. כל תהליך קפוא מחזיק לפחות משאב אחד שחילוצו היה מאפשר לו לצאת מהקיפאון. לעומת זאת, במקרה הכללי של הרעבה, חסימת התהליך אינה קשורה כלל לפעולותיו בעבר. משאב חיוני נמנע ממנו ללא כל קשר למעשיו. הסיבה לכך היא מדיניות הקצאת המשאבים של המערכת. היא זו שצריכה להחליט מי התהליך הבא שיקבל את המשאב, ומדיניות זו עלולה להוביל להרעבה. הקצאת משאבים בהתאם לעדיפויות קבועות היא גורם אופייני להרעבה. תהליך בעל עדיפות נמוכה צפוי להרעבה.

דרך מקובלת להימנע ממלכודת זו היא על ידי הוספת מנגנון הזדקנות המבטיח כי תהליך הממתין זמן רב – עדיפותו עולה. התור הרגיל, לעומת זאת, הוא מדיניות הקצאת משאבים החסינה מפני הרעבה. ברגע שהתהליך נכנס לסוף התור, כבר ידוע בבירור כי ככל שיחלוף הזמן הוא יתקדם לכיוון ראש התור.

פרק 7 – UNIX סיכומים

עקרונות UNIX

מערכת TIMESHARING אינטראקטיבית. עוצבה על ידי מתכנתים למתכנתים, מאפשרת למספר אנשים לעבוד ביחד ולחלוק אינפורמציה בצורה מבוקרת. המערכת צריכה להיות פשוטה אלגנטית ועקבית.

דוגמא לעקביות – אם הפקודה LS A* מציגה את כל הקבצים שמתחילים ב- A, אזי הפקודה RM A* תמחק את כל הקבצים שמתחילים ב- A. תכונה זו נקראת PRINCIPLE OF LEAST SURPRISE.

דבר נוסף שמתכנתים דורשים הוא POWER AND FLEXIBILITY. כלומר שלמערכת יהיו מספר קטן של אלמנטים בסיסיים שיבצעו את הצרכים של האפליקציה.

אחד הדברים שמובנים ב-UNIX הוא שכל תוכנית תעשה רק דבר אחד, אך תעשה אותו היטב.

כמו כן פקודות קצרות – כלומר מספיק CP ולא צריך COPY, ולא להסביר כל פעולה – דרושה מערכת משרתת לא אומנת.

הממשק של UNIX

UNIX בנויה כפירמידה, בסיסה הוא החומרה, לאחר מכן מערכת ההפעלה (ניהול תהליכים, זיכרון, קבצים, פלט\קלט ועוד) שתפקידה לשלוט על החומרה ולספק קריאות מערכת לכל התוכניות, דבר המאפשר למשתמש\תוכניתן ליצור ולנהל תהליכים, קבצים ושאר משאבים.

התוכניתן יוצר קריאות מערכת על ידי השמת ארגומנטים באוגרים או במחסנית, ואלו מאפשרים מעבר מ- USER MODE אל KERNEL MODE.

מה שלא ניתן לכתוב ב-C מסופק כספריות של מערכת ההפעלה באסמבלר. (זוהי השכבה השלישית – STANDARD LIBRARY מכילה את OPEN, CLOSE, READ,WRITE, FORK ועוד). אך ניתנת לקריאה מתוכנית ב-C. הקריאה מהתוכנית היא לספריה, (תוך תאור הפרמטרים, מה היא עושה ומה התוצאות הדרושות) ולא לקריאת המערכת עצמה.

בשכבה הרביעית יש תוכניות ייעודיות כגון המעטפת – SHELL, מעבדי תמלילים מהדרים וכו'. ובשכבה האחרונה ממשק המשתמש.

ב- USER MODE יש שלושה ממשקים: SYSTEM CALL INTERFACE, LIBRARY INTERFACE, USER INTERFACE.

ניתן לשנות את ממשק משתמש בלי לשנות דבר במערכת ההפעלה.

כניסה ל-UNIX

זהו תהליך חובה, כל משתמש נכנס עם שם משתמש וסיסמא ועל פיהם מוגדרות לו הרשאות הגישה לקבצים – התהליך מבוצע על ידי תוכנית ה- LOGIN. ברוב המערכות לשיתוף זמן קבצי המשתמשים והסיסמאות חבויים. ב-UNEX לא, הקובץ מכיל שורה עבור כל משתמש – שם משתמש, מספר ID, סיסמא מוצפנת, ספרית עבודה ועוד אינפורמציה. בכניסה מושווה לשורה המתאימה בקובץ. היופי במנגנון הוא שכל תוכנית יכולה לבדוק מי הבעלים של קובץ – מה ה- ID שלו ולהשוות עם קובץ הסיסמאות.

UNIX SHELL

תוכנית זו רצה לאחר ה- LOGIN ומדפיסה למשתמש סמן של שורת פקודה וממתינה לפקודה אותה יקליד המשתמש. כשהמשתמש מקיש פקודה המעטפת גוזרת את המילה הראשונה בפקודה ומחפשת תוכנית בשם זה. אם נמצאה, התוכנית מורצת והמעטפת משהה את עצמה עד לסיום התוכנית. המעטפת היא בעצם תוכנית משתמש שכל מה שהיא יודעת לעשות היא לקרוא ולכתוב מטרמינל ולהריץ תוכניות אחרות.

ארגומנטים ששולטים בפעולה של פקודה נקראים FLAG לדוגמא .HEAD –20 FILE

להדפיס את 20 השורות הראשונות בקובץ ה- 20 הוא FLAG.
כדי להקל את העבודה מול קבצים רבים ארגומנטים ודגלים, המעטפת מכירה תווים מיוחדים הנקראים WILDCARDS כגון *ץ או ?.
רוב התוכניות משתמשות בשלושה קבצי STNDARD : INPUT, ERROR, OUTPUT.
באופן נורמלי הקלט הוא מהמקלדת והשניים האחרים שולחים נתונים למסך. ניתן לשנות זאת בצורה הבאה : >OUT מפנה לקובץ OUT. INT< מפנה לקובץ IN.
הסימן | מעביר מידע בין שני פקודות למשל SORT <IN | HEAD –30 הסימן נקרא צינור. הדוגמא הוא לוקח את הפלט של SORT ומשתמש בו כקלט ל-HEAD .
אוסף פקודות המחוברות על ידי צינור ניקרא PIPELINE. אין כמעט מגבלות למה שאפשר ליצור על ידי שרשור פקודות.
UNIX אינה רק מערכת שיתוף זמנים אלא גם MULTIPROGRAMING SYSTEM.
משתמש אחד יכול להפעיל מספר תוכניות באותו זמן. כל אחד בתהליך נפרד.
על מנת להריץ תהליך ברקע מסיימים את הפקודה עם &. כנ"ל גם לגבי PIPELINES.
קובץ שמכיל פקודות מעטפת נקרא SHELL SCRIPTS, יכול להכיל גם תנאים, לולאות וכדומה (אחת האפשרויות ליצור זאת הוא על ידי BERKELEY C SHELL).

קבצים וספריות ב- UNIX

המערכת מתייחסת לקבצים כאוסף של בתים. לא אכפת לה כלל מה תוכן הקבצים ולמה הם משויכים. שם קובץ הוא 14 תווים, אולם בחלק מהגרסאות שונה ל- 255 תווים כמו ב-BERKELEY.
קובץ מוגן על ידי 9 ביטים שנקראים RIGHTS BITS. שלושת הראשונים מגדירים מיהו הבעלים. שלושת הבאים מייצגים את הקבוצה של הבעלים, שלושת האחרונים מוקצים לשאר.
הקבצים מרוכזים תחת ספריות המאוחסנות בדיוק כמו קבצים, במקום הרשאות הרצה, לקובץ יש הרשאות חיפוש, עיון בספריה.

תוכניות עזר ב-UNIX

ממשק משתמש מורכב לא רק ממעטפת אלא גם מתוכניות עזר רבות שניתן לחלקן ל- 6 קטגוריות עיקריות :

1. פקודות לניהול קבצים וספריות.
2. FILTERS.
3. COMPILERS AND PROGRAM DEVELOPMENT TOOLS.
4. TEXT PROCESSING.
5. SYSTEM ADMINISTRATION.
6. שונות.

שלושת הקטגוריות הראשונות מאפשרות לכל אחד לכתוב מעטפת על ידי שימוש בתבניות הנמצאות בהם.
בכלי תכנות וקומפילציה ישנם הכלי CC הקורא לקומפיילר של C , ו- AR האוסף פרוצדורות ספריה אל קובצי הארכיב. כמו כן יש את MAKE על מנת ליצור תוכנית גדולה המורכבת ממספר קבצי מקור. כמו כן שומר על הקשר בן קובצי התוכנית לקובצי הכותרת – HEADER.

FUNDAMENTAL CONCEPTS IN UNIX

תהליכים ב- UNIX

היחידה הפעילה ב-UNIX היא התהליך. כל תהליך מריץ תוכנית יחידה. מאחר שכל אחד יכול להריץ מספר תהליכים, אזי יתכן מצב בו ירוצו תהליכים רבים במקביל, אפילו בזמן

שהמשתמש לא מריץ דבר, רצים תהליכי מערכת הנקראים "שדים". תהליכים אלו מתחילים לרוץ עם עליית המערכת.
"שד" טיפוסי הוא CROM שרץ פעם בדקה, בודק אם יש עבורו עבודה, אם כן מבצע אותה וחוזר לישון, אחרת חוזר לישון עד הפעם הבאה שירוץ.
CROM טוב לתזמן זמנים, לשליחת עבודות לשעה מסוימת וכדומה. "שדים" אחרים אחראיים על דואר אלקטרוני, ניהול תור של מדפסת, בדיקה האם יש מספר דפים פנויים בזיכרון.
לתהליך שממנו נוצר תהליך אחר קוראים תהליך אב, לתהליך החדש קוראים בן, כל אחד מהם מקבל סביבת זיכרון משלו. אם יש שינוי במשתנים בתהליך האב, הדבר לא ישפיע על תהליך הבן, ולהפך. קבצים פתוחים ניתנים לשיתוף בין האב לבן. אם קובץ היה פתוח אצל ההורה לפני קריאת FORK, הוא ימשיך להיות פתוח עבור הבן וההורה לאחר מכן.
על מנת להבדיל בין תהליך האב לתהליך הבן FORK מחזיר לתהליך הבן את המספר 0 ולתהליך האב מספר השונה מ-0 ב- PID. אם תהליך הבן רוצה לדעת מה ה- PID שלו הוא מפעיל את קריאת המערכת GETPID.
בסופו של דבר מאחר שיכולים להיות הרבה ילדים וגם להם ילדים נבנה עץ של תהליכים.
עם עליית המערכת עולה התהליך INIT. תהליך זה קורא מספריית /ETC/TTYS מה מספר המסופים\כניסות שיש למערכת. ואינפורמציה לגבי כל אחת מהכניסות. לאחר מכן משאיר בן בכל כניסה, שמפעיל את תוכנית LOGIN, והולך לישון עד שאחד המסופים ינסה להיכנס לעבודה. בכל מסוף מופיע LOGIN , לאחר ה- LOGIN נתבקש להקיש סיסמא, הסיסמא תיבדק בספריה /ETC/PASSWD
אם הסיסמא נכונה, ה- LOGIN יפעיל את מעטפת המשתמש.
ניתן ליצור קשר בין תהליכים, לקשר זה קוראים PIPE. ניתן לסנכרן בין התהליכים משום שאם תהליך מנסה לקרוא מצינור שלא הגיע ממנו מידע אזי הוא יחסם עד שהמידע יהיה זמין.
דרך נוספת ליצור קשר בין תהליכים הוא פסיקות תוכנה. תהליך יכול לשלוח לתהליך אחר SIGNAL, התהליך המקבל יכול להתעלם או לתפוס אותו ולהגדיר מה לעשות אתו, או להרוג את התהליך (ברירת מחדל).
אם הוחלט לתפוס אותו, אזי חייב להגדיר פרוצדורה שנקראת SIGNAL HANDLING. לאחר סיום הטיפול בסיגנל התהליך יחזור לנקודה אחרי הנקודה בה הוא פנה לפונקציית הטיפול בפסיקות.
תהליך יכול לשלוח סיגנלים רק לתהליכים הנמצאים יחד עמו ב- GROUP PROCESS שכוללים את הוריו וילדיו. סיגנלים משמשים גם לטיפול בנקודה צפה, לטיפול בחילוק ב-0.
לכל משתמש יש UID הרשום ב- PASSWORD FILE, כל תהליך משויך ל- UID של האדם שיצר אותו.
משתמש אב נקרא SUPERUSER או ROOT, משתמש זה יכול לקרוא ולכתוב לכל הקבצים, ויכול ליצור קריאות מערכת שמשתמשים אחרים לא יכולים לגשת אליהם.
בעמוד 284 יש רשימה של הסיגנלים הדרושים ל- POSIX.
ב- UNIX יש ביט המשויך לכל תוכנית רצה הנקרא SETUID BIT. ביט זה הוא חלק מהביטים המוקצים ל- PROTECTION MODE.

זיכרון ב- UNIX

כל תהליך ב- UNIX מקבל מרחב כתובות המחולק לשלושה סגמנטים: TEXT, DATA, STACK.
TEXT – אלו הפקודות שהתוכנית צריכה לבצע, כתוב באסמבלר לאחר המרה שעברה התוכנית המקורית, באופן רגיל סגמנט זה הוא בעל הרשאות לקריאה בלבד.
DATA – מכיל משתני תוכנית, מחרוזות, מערכים, ושאר המידע של התהליך. חלק מה-DATA מחולק ל- 2: DATA ו- BBS. ב- DATA יש את המידע הקיים כבר עם עלית

התהליך. ה- BBS הוא מקום שמור למידע שאנו מצפים לו (משתנים שלא אותחלו וכדומה) אך עדיין אין לנו את ערכיו. החלק של ה- DATA (כולו) גדל וקטן במהלך התוכנית על ידי קריאות מערכת (בקריאות אלו נעשה שימוש רק על ידי הפונקציה MALLOC בשפת C).

STACK – בהרבה מכונות מתחיל מהחלק העליון של הזיכרון המדומה ויורד עד 0. אם יורד מתחת לכך תתרחש HARDWARE FAULT, ומערכת ההפעלה תוריד את תחתית המחסנית בדף אחד. כשתוכנית רצה, המחסנית אינה מתחילה מ- 0, אלא מכילה את מאפייני המעטפת ואת הפקודות שהוקשו במעטפת.

ניתן לחלוק את סגמנט ה- TEXT בין תוכניות – לדוגמא, עורך, אולם ה- DATA וה- STACK לעולם לא יהיו משותפים. אם הם צריכים לגדול ואין מקום עבורם הם יועברו למקום אחר בו יש מקום בזיכרון.

<u>מערכת הקבצים ב- UNIX</u>

כדי לפתוח קובץ משתמשים בקריאת המערכת OPEN שהפרמטרים שלה הם המסלול בו נמצא הקובץ, והקובץ. באם הקובץ קיים ויש לבקש הרשאות מוחזר מספר שלם שנקרא FILE DESCRIPTOR לקורא, אחרת, אם לא קיים או אין הרשאות, מוחזר 1 -.

הקריאות לכתיבה או קריאה מהקובץ משתמשות ב- FILE DESCRIPTOR על מנת לזהות את הקובץ.

כאשר התהליך עולה יש לו שלושה FILE DESCRIPTORS לשימושו: 0 עבור הקלט הסטנדרטי, 1 עבור הפלט הסטנדרטי, ו- 2 עבור שגיאות דאנדרטי.

הקובץ הראשון שיפתח יקבל את הערך 3 וכך הלאה בסדר רץ. כשסוגרים קובץ מתפנה מספר וניתן להשתמש בו לקובץ אחר.

ABSOLUTE PATH – המסלול המלא מהשורש ועד למיקום הקובץ המבוקש.

RELATIVE PATH – המסלול מהמקום בו אנו נמצאים כרגע אל הקובץ המבוקש.

LINK - ניתן להצביע לקובץ הממוקם בספריה אחת מספריה אחרת. הדבר מתבצע על ידי LINK ונקרא LINKED FILE.

באם יש לנו מערכות קבצים נפרדות, ניתן לבצע MOUNT לאחת מהן, כך שהיא תראה לנו כחלק מהעץ של המערכת השניה (תאור בעמוד 289 איור 10-7).

מאפיין נוסף למערכת הקבצים הוא LOCKING . ניתן לסמן קובץ בצורות הבאות:

SHARED ואז גם משתמשים נוספים שנגשים לאותו קובץ יכולים להשתמש בו, או EXCLUSIVE ואז רק אחד יכול להשתמש בקובץ עד שהוא משחרר את הנעילה. באם מסומן SHARED לא יכולים המשתמשים הבאים לאחר המסמן, ולפני שהוא יצא, לשנות את המאפיין ל- EXCLUSIVE. (לא חייב שהנעילה תהיה על קובץ שלם, יתכן שרק על חלקים ממנו).

7.3.4 <u>קלט\פלט ב-UNIX</u>

כמו כל המחשבים, אלו שמריצים UNIX מכילים התקני קלט\פלט כגון מסופים, דיסקים, מדפסות ורשתות תקשורת המחוברות אליהם.

UNIX מתקשר עם אמצעי קלט\פלט אלו על מה שנקרא SPECIAL FILES. לרוב נמצאים במסלול /dev. לדוגמא מדפסת תהיה /dev/lp. מסוף 1 /dev/tty1 ורשת /dev/net. הגישה לקבצים אלו היא כמו ליתר הקבצים. ניתן להגביל גישה לאמצעי קלט\פלט בדיוק כמו לקבצים על ידי הרשאות מתאימות לספריית /dev.

קבצי SPECIAL FILES מחולקים לשתי קטגוריות:

א. BLOCK SPECIAL FILES - כאשר ניתן לגשת ישירות לבלוק מבוקש בלי לעבור על כל אלו שלפניו.

ב. CHARACTER SPECIAL FILES – עבור התקנים שעובדים בצורה של תווים, כגון מסופים, מדפסות ורשת.

ה-SOCKET מקביל לתא דואר, או לטלפון, והוא יוצר FILE DESCRIPTOR הדרוש ליצירת הקשר, קריאה וכתיבה של מידע ושחרור הקשר. תומך בהעברה של בתים, בהעברה של PACKETS. האפשרות השלישית היא UNRELIABLE PACKET TRANSMISSION. משמשת לאפליקציות זמן אמת ועבור סיטואציות שהמשתמש רוצה בקרת שגיאות.
בשיטה הראשונה מאפשרים לשני תהליכים בשני מחשבים שונים להקים צינור ביניהם כאשר ביטים מוכנסים מצד אחד ויוצאים בשני. השיטה השניה דומה לראשונה אולם אם השולח מבצע 5 קריאות WRITE נפרדות, כל אחת עבור 512 בתים, והמקבל מבקש 2560 ביטים, עם SOCKET מהסוג הראשון, כל 2560 הבתים יחזרו בבת אחת. בטיפוס השני רק 512 בתים יחזרו. יש צורך ב- 4 קריאות נוספות לקבל את השאר. בסוג השלישי משתמשים כדי לתת למשתמש כניסה לרשת, והוא שימושי במיוחד עבור אפליקציות זמן אמת. כל SOCKET מקבל כתובת לפני שליחתו (לרוב 32 ביט). יש שימוש בקריאת מערכת LISTEN שיוצר חוצץ ובלוק עד שהנתונים מגיעים, וקריאת מערכת CONNECT הנותנת פרמטרים של FILE DESCRIPTOR עבור ה- SOCKET המקומי ואת הכתובת של REMOTE SOCKET. במהלך הקשר התקשורת מתפקדת כמו צינור.

7.4 קריאות מערכת ב-UNIX

כשקריאת מערכת מחזירה 0 היא הצליחה, -1 נכשלה. כשיש שגיאה קוד השגיאה מוחזר במשתנה בשם errno, תמיד צריך לבדוק האם יש שגיאה.

ניהול קריאות מערכת

1. FORK – הדרך היחידה ליצור תהליך חדש. יוצר העתק של התהליך המקורי. כולל אורגים משתנים וכו', מרגע זה כל אחד פונה לדרכו, ושינוי באחד לא ישפיע על השני. מחזירה 0 לילד ומספר כלשהו לתהליך האב. לאחר מכן, לרוב, תהליך האב צריך להמתין לתהליך הבן, פעולה זו מתבצעת על ידי הקריאה WAITPID שממתינה עד שתהליכי הבנים יסתיימו. היא מקבלת שלושה פרמטרים, הראשון אומר לאיזה ילד לחכות, אם –1 הילד הראשון, הפרמטר השני מקבל את סטטוס הסיום של הילד: סיום תקין או לא.
2. EXEC – על מנת להריץ פקודות קיימת קריאת המערכת EXEC. לרוב עם 3 פרמטרים, שם הקובץ שרוצים להריץ, מצביע למערך של פרמטרים, ומצביע למערך של סביבה.
3. SIG-ACTION – על מנת להפעיל סיגנלים. הפרמטר הראשון הוא הסיגנל אותו רוצים לתפוס, השני הוא מצביע למבנה של מצביעים לפרוצדורות לטיפול בסיגנלים, השלישי הוא מצביע למבנה אליו תוחזר אינפורמציה לגבי ביצוע הסיגנלים. לאחר הפעלת הסיגנל תוחזר השליטה לנקודה בה בוצע הסיגנל. SIG-ACTION משמש גם להתעלמות מסיגנלים ולהריגת תהליכים.
4. KILL – קריאת המערכת מאפשרת לתהליך אחד לשלוח סיגנל לתהליך אחר.
5. ALARM – על מנת להפסיק פעולה אחרי פרק זמן מסוים על ידי הסיגנל SIGALRM.
6. PAUSE – משהה תהליך עד שיגיע סיגנל.

קריאות מערכת לניהול זיכרון

אין קריאות מערכת לניהול זיכרון, למעט קריאה בשם BRK שאיתה ניתן להגדיר את גודל הסגמנט.

קריאות מערכת לקבצים וספריות

CREATE - על מנת ליצור קובץ. הפרמטרים הם שם הקובץ, וההגנות שלו. בצורה הבאה: fd = creat ("abc", mode); הקובץ נפתח לכתיבה, באם הפעולה מבוצעת לקובץ קיים, אזי גודלו משתנה ל- 0 (תלוי בהרשאות של הפותח).
לפתוח קובץ קיים לקריאה או כתיבה, הפרמטרים הם שם הקובץ והאם הפתיחה לצורכי כתיבה, קריאה או שניהם.

CLOSE - על מנת לסגור קובץ.
LSEEK – מצביע על הבית הבא שיקרא או ייכתב. טוב עבור הקריאות READ, WRITE. ל- LSEEK שלושה פרמטרים, שם הקובץ, המיקום, השלישי אומר האם בתחילת הקובץ, בתוכו או בסופו. הערך המוחזר הוא מיקומו המוחלט של הקובץ.
STAT - ניתן לקבל מאפיינים על קובץ – רשימת המאפיינים בעמוד 298 שרטוט 15-7. הפרמטר הראשון הוא שם הקובץ והשני הוא מצביע למבנה בו אנו רוצים לקבל את האינפורמציה.
MKDIR - ליצירת ספריה.
RMDIR – להסרת ספריה, רק אם היא ריקה.
LINK – לקובץ יוצר כניסה נוספת לספריה, ניתן להסיר רק על ידי UNLINK .
כאשר מסירים מקובץ את כל הLINKS הוא נימחק.
CHDIR – משנה ספריה.
CHMOD – משנה הגנות של קובץ.

<u>קריאות מערכת עבור קלט\פלט</u>
לא לדיון בקורס זה.

7.5 IMPLEMENTATION OF UNIX

MACHINE-DEPENDENT KERNEL – חלק מהגרעין. הקוד מורכב ממתאם פסיקות עבור התקני, ומתאמי קלט\פלט, וחלק מתוכנת ניהול הזיכרון. הרוב כתוב ב- C, אך מאחר שפונה ישירות לחומרה, צריך לשנות את הקוד ממכונה למכונה.
MACHINE-INDEPENDENT KERNEL – לעומת זאת זהה בכל המכונות מאחר שלא מתייחס לחומרה. כולל קריאות מערכת, ניהול תהליכים, תיזמון, צינורות, סיגנלים, דפדוף, מערכת הקבצים ואת החלק העליון של מערכת קלט\פלט. חלק זה גדול מהחלק השני.

<u>יישום תהליכים</u>
לכל תהליך יש חלק שנקרא USER וחלק שנקרא KERNEL. חלק הגרעין מתחיל לפעול רק שמתרחשת קריאת מערכת. לגרעין יש מחסנית ומונה, והוא מכיל שני מבנים השייכים לתהליך, PROCESS TABLE, USER STRUCTURE.
טבלת התהליכים קיימת כל הזמן ומכילה אינפורמציה על כל התהליכים, כולל אלו שלא נמצאים כרגע בזיכרון.
USER STRULTURE – אם התהליך אליו שייך המבנה לא נמצא בזיכרון, מוצא המבנה מהזיכרון לדיסק.
טבלת התהליכים מכילה את הקטגוריות הבאות:

א. SCHEDULING PARAMETERS – עדיפות, זמן CPU זמן שינה.
ב. MEMORY IMAGE – מצביעים ל-TEXT , DATA, STACK SEGMENT. או אם משתמשים בדפדוף אל טבלת הדפים. אם סגמנט הטקסט משותף אזי יש מצביע לטבלת הטקסט המשותפת. כאשר תהליך לא נמצא בזיכרון אזי יש מידע היכן ניתן למצוא אותו.
ג. סיגנלים – אלו סיגנלים התקבלו, אלו חסומים, מאלו מתעלמים ואלו נשלחו.
ד. MISCELLANEOUS , שונות – מצב נוכחי של תהליך, אירועים שממתינים להם אם יש, הזמן הנותר לסיום, PID, PID של הורה, זיהוי המשתמש וקבוצת המשתמש.

USER STRUCTURE מכיל אינפורמציה שלא זקוקים לה כשהתהליך לא בזיכרון.
המידע הנמצא ב- USER STRUCTURE הוא:

א. רגיסטרים – שמירת ערך האוגרים.
ב. מצב קריאת מערכת – מידע על קריאות המערכת כולל פרמטרים ותוצאה.

ג. FILE DESCRIPTOR TABLE – כאשר מתרחשת קריאת מערכת, מתקבל מידע לגבי קריאת המערכת המשמש כאינדקס לטבלה הזו, וזאת כדי לאחסן את ה- I-NODE הקשור לקובץ.

ד. ACCOUNTING – מצביע לטבלה המכילה מידע על המשתמש וזמן CPU של תהליך.

ה. מחסנית הגרעין – מכיל מידע לשימוש הגרעין של התהליך.

כאשר FORE מופעל נעשית פניה לגרעין על מנת שימצא מקום פנוי בטבלת התהליכים, אם מוצא אזי מעתיק את כל האינפורמציה של ההורה לכניסה של הבן. לאחר מכן מקצה זיכרון לתהליך הבן מעתיק את הסגמנטים של האב. ואז תהליך הבן מוכן לרוץ.
תהליך חלוקת הזמנים נעשה בשתי רמות. ברמה הנמוכה כל תהליך בזיכרון מקבל זמן לרוץ, ברמה הגבוהה מבוצע מעבר מהדיסק לזיכרון וחזרה על מנת שלכולם תהליך תהיה הזדמנות לרוץ.

ברמה הנמוכה יש מספר תורים עם עדיפויות. תהליך שרץ ב- USER MODE מקבל ערך חיובי, תהליך בגרעין מקבל ערך שלילי. ערך שלילי בעל עדיפות גבוהה יותר. רק תהליכים בזיכרון יכולים לרוץ. תהליך יכול לרוץ יחידת זמן אחת, או עד שהוא נחסם. בכל CLOCK מורדת עדיפות לתהליך ב- 1. ואז הוא עובר לתור בעל עדיפות נמוכה יותר.
כשהתהליך מסיים את הזמן המוקצה לו הוא מועבר לסוף התור, צורה זו נקראת ROUND ROBIN ALGORITHM.
הדרך השניה היא: כל מוני השימוש במעבד מחולקים ב- 2 על מנת למנוע מתהליך לדרוש כל הזמן שימוש מהיר במעבד. לאחר מכן מחושבות העדיפויות מחדש לפי הנוסחה:
NEW PRIORITY = BASE+CPU USAGE. הבסיס בדרך כלל שואף ל- 0, אולם ניתן לשנות אותו לערך גבוה יותר (לדוגמא על ידי NICE).
משתמשים המחכים לקלט\פלט או למידע האמור להגיע מאמצעים אחרים יחכו במתחם הגרעין כשהעדיפות שלהם תהיה שלילית. הרעיון בלתת לתהליך הממתין לאמצעי קלט\פלט עדיפות גבוהה היא המתבזבז המון זמן בהמתנה לקלט\פלט לכן ברגע שהוא יכול לעבוד (קיבל את הקלט) אזי צריך להכניסו מיד לעבודה.

<u>ניהול זיכרון ב-UNIX</u>

UNIX עובדת בשיטה של SWAPPING, כאשר תהליך לא יכול להישמר בזיכרון הוא מועבר לדיסק.

<u>SWAPPING</u>

התנועה בין הזיכרון לדיסק מנוהלת על ידיד הרמה הגבוהה יותר של מנגנון התזמון (SCHDELER) הנקרא SWAPPER. המעבר מהזיכרון לדיסק נעשה כאשר הגרעין לא מוצא מקום בזיכרון, וזה קורה במקרים הבאים:

א. FORK צריך מקום בזיכרון לתהליך בן.

ב. BRK – כאשר קריאת מערכת זו צריכה להגדיל את סגמנט הנתונים.

ג. כאשר המחסנית גודלת מעבר לשטח שהוקצה לה.

כשנדרש לפנות מקום ה- SWAPPER מחפש תהליכים הממתינים למשהו (קלט\פלט וכדומה). באם נמצא רק אחד מוריד אותו, באם יותר אזי עושה תחשיב של עדיפות + זמן בזיכרון, בעל הערך הגבוה ביותר נבחר (מאחר שכבר צרך הרבה זמן CPU וזיכרון). באם אין תהליכים חסומים הממתינים למשהו, אזי יופעל אותו אלגוריתם על תהליכים המוכנים להרצה (מצב READY). כל מספר דקות ה- SWAPPER בונה רשימה של תהליכים הנמצאים בדיסק ומוכנים להרצה, התהליך המתין הכי הרבה זמן בדיסק יבחר, כמו כן בודק אם תהיה החלפה קלה או החלפה קשה.
החלפה קלה – היא החלפה שבה יש מספיק זיכרון להעלות את התהליך בלי להוריד תהליכים אחרים.
החלפה קשה – נדרש להוריד תהליך או יותר מהזיכרון על מנת לטעון תהליך מהדיסק. בהחלפה קשה נדרש קודם לפנות מקום ואז להעלות את התהליך.
אלגוריתם זה נמשך עד שאחד מהתנאים הבאים מתקיים:

1. אף תהליך על הדיסק לא מוכן להרצה.

2. אין מקום בזיכרון לשום תהליך נוסף.

מקום פנוי בזיכרון נשמר כרשימה מקושרת של החורים בזיכרון, כנ"ל לגבי ה- SWAP DEVICE. כשצריך זיכרון רצים על רשימת החורים עד שמוצאים את החור הראשון שמספיק גדול, מקצים אותו ומשחררים מהרשימה (מודל זה מיושם כמעט בכל UNIX).

<u>דפדוף</u>

התהליך לא חייב להיות בזיכרון כדי לרוץ, כל מה שצריך להיות בזיכרון הוא USER STRUCTURE וטבלת הדפדוף. אם אלו בזיכרון אזי התהליך נרשם בזיכרון ולכן יכול לרוץ. הדפים של הנתונים, הטקסט, והמחסנית דינמיים. אם ה- USER STRUCTURE וטבלת הדפדוף של התהליך לא בזיכרון התהליך לא יכול לרוץ.

מנגנון הדפדוף מנוהל על ידי שד הדפים. שד הדפים בודק אם מספר הדפים הפנויים בזיכרון נמוך מידי, ואז פועל כדי לשחרר דפים נוספים לזיכרון. מספר תהליך השד הוא 2.

תהליך 0 הוא ה- SWAPPER, ותהליך מספר 1 הוא INIT. הזיכרון הראשי ב- 4BSD מחולק לשלושה חלקים. שני הראשונים, KERNEL ו- CORE MAP נמצאים בחלק התחתון ולעולם לא יפונו מהזיכרון. שאר הזיכרון יחולק למסגרות דפים, כך שכל מסגרת יכולה להכיל טקסט, נתונים ומחסנית או טבלת דפים. או להיות ריקה\פנויה.

ה- CORE MAP מכילה מידע על תכולת המסגרות. פחות מ- 2% מהזיכרון מנוצלים על ידי ה- CORE MAP. כניסה 0 במפר מתארת את מסגרת 0, כניסה 1 את מסגרת 1 וכך הלאה. עם 16 מסגרות פחות מ- 2% מהזיכרון מנוצלים על ידי ה- CORE MAP. שתי הכניסות הראשונות בכניסה במפה (איור 17-7 בעמוד 305) בשימוש כאשר המסגרת נמצאת ברשימת הפנויים. הכניסה הראשונה מצביעה לבה בתור ברשימה, והשניה לקודם. שלושת הכניסות הבאות מגדירות את מיקום הדף בדיסק כאשר הוא מורד לדיסק. שלושת הבאות מגדירות את הכניסה בטבלת התהליכים, המחסנית והמיקום במחסנית.

באם לא כל הדפים של תהליך נמצאים בזיכרון, אזי נקבל PAGE FAULT ואז המערכת תחפש ברשימת הפנויים דף פנוי ותטען לתוכו את הדף החסר, אם הרשימה של הפנויים ריקה, אזי התהליך יושהה עד שיתפנו דפים.

<u>אלגוריתם להחלפת דפים</u>

החלפת דפים נעשית על ידי שד הדפים. כל 250 מילי-שניות הוא מתעורר ובודק האם מספר הדפים נמוך או שווה לפרמטר של המערכת – הנקרא LOTSFREE (בדרך כלל מכוון לרבע מהזיכרון). אם פחות אזי מתחיל להעביר דפים לדיסק עד למצב בו יש מספיק דפים פנויים. אם מהתחלה יש מספיק דפים פנויים, השד חוזר לישון. אם יש מעט תהליכים והרבה זיכרון פנוי, שד זה ישן רוב הזמן.

ב- 4BSD המנגנון עובד בצורה הבאה: אם יש תהליך שנח 20 שניות הוא מורד מהזיכרון על ידי ה-SWAPPER , באם אין אזי נבדקים 4 התהליכים הגדולים ביותר, זה שנמצא הכי הרבה זמן בזיכרון יורד. התהליך התואר קודם קורה באם הדפדוף לא מצליח להגיע למספיק מקום כפי שנדרש על ידי ה- LOTSFREE.

ב- SYSTEM V משתמשים לפי אלגוריתם ONE-HANDED CLOCK כלומר מפנה תהליך מהזיכרון אם לא היה בשימוש N מעברים רצופים. בשאר המערכות N=2. כלומר פינוי לאחר 2 מעברים.

במקום המשתנה LOTSFREE ב- SYSTER V מוחזקים שני משתנים MAX MIN על מנת למנוע מצב שהעלאה של תהליך לזיכרון תביא אותנו למצב שאנו שוב מתחת ל- LOTSFREE.

<u>מערכות קבצים ב- UNIX</u>

בלוק 0 בכל דיסק ב-UNIX לא בשימוש ומכיל קוד עבור BOOT. בלוק מספר 1 נקרא SUPERBLOCK, ומכיל מידע קריטי על סידור הדיסק כמו מספר I-NODE, ותחילת הרשימה של מיקום פנוי הדיסק. כל I-NODE בגודל 64 בתים ומתאם קובץ אחד, מכיל

מידע על הבעלים וההגנה של הקובץ. בסך הכל מכילה טבלת I-NODE מספיק מידע על מנת להגיע לכל המידע בדיסק.
חוץ מ- I-NODE יש DATA BLOCK שמכיל בעצם את כל הקבצים והספריות. I-NODE מספר 1 שמור ל- BAD BLOCK HANDLING. I-NODE 2 עבור הספרייה הראשית, /.
שמירת המיקום של תהליך בקובץ, לא תעשה בטבלת ה- I-NODE, מאחר שאם יהיה יותר ממשתמש אחד בקובץ, אזי לא ניתן לשמור את המיקום של כל אחד מהם.
ה-I-NODE מכיל את כתובת 10 הבלוקים הראשונים של הקובץ. אם המיקום בקובץ הוא ב- 10 הבלוקים הראשונים הללו אזי קוראים מהבלוק הדרוש והנתונים עוברים למשתמש.
עבור קבצים ארוכים מ- 10 בלוקים יש שדה ב- I-NODE המכיל את הכתובת בדיסק של SINGLE INDEREC BLOCK (איור 19-7 עמוד 310). הבלוק מכיל את הכתובות בדיסק של עוד בלוקים הקשורים לקובץ. DOUBLE INDIRECT BLOCK – מכיל את הכתובות של 256 SINGLE INDEREC BLOCKS שכל אחד מהם יכול להכיל כתובות של 256 DATA BLOCKS. אם זה לא מספיק יש ב- I-NODE מקום עבור TRIPLE INDIRECT BLOCK.
ספריה מכילה 16 בתים של כניסות כאשר כל כניסה מכילה שם קובץ עד 14 תווים ומספר I-NODE של הקובץ.

THE BERKELEY FAST FILE SYSTEM

בגירסה זו שם יכול להכיל עד 255 תווים, אין את מבנה 16 הבתים של כניסות. על ידי OPENDIR, CLOSEDIR, READDIR ניתן לגשת מתוכנית לספריות מבלי לדעת כיצד בדיוק הם ממומשים.
שינוי נוסף הוא ריכוז חלקי הקובץ על אותו הצילינדר על מנת לאפשר גישה מהירה.
שינוי שלישי הוא שמירת שני גדלים של גודל בלוק, על מנת לאפשר חסכון במקום כשמדובר בקבצים קטנים. החיסרון הוא שיותר קשה לנהל את הדיסק.

קלט\פלט ב- UNIX

התקני הקלט\פלט מנוהלים על ידי מתאם ההתקן, עבור כל אמצעי יהיה לפחות מנהל התקן אחד. המתאם מקשר את האמצעי למערכת ההפעלה.
אמצעי הקלט\פלט מתחלקים בגדול לשניים, כאלו שעובדים בצורה של בלוקים וכאלו שעובדים בצורה של תווים.
על מנת להמעיט בהעברה של בלוקים הלוך וחזור ישנו BUFFER CACHE בין DISCK DRIVER לבין מערכת הקבצים. שם מוחזקים הבלוקים השימושיים ביותר. בעת פניה לבלוק שלא נמצא בזיכרון, ייבדק אם אינו נמצא בחוצץ, אם נמצא אזי יועבר משם לזיכרון, ונחסכה פניה לדיסק. (איור 20-7 עמוד 311). אם לא נמצא, יועתק מהדיסק לחוצץ וממנו למקום בו זקוקים לו. כדי לנהל את החוצץ, מוחזקת רשימה מקושרת, כאשר נגשים לבלוק מסוים הוא מועבר לראש הרשימה, כאשר צריכים לפנות מקום מורידים מסוף הרשימה. כאשר צריכים לכתוב לדיסק כותבים לחוצץ, ורק כשאין ברירה (צריך לפנות מקום או עבר פרק זמן מוגדר) מעתיקים את המידע לדיסק – שוב על מנת לחסוך בגישות לדיסק.
מידע בתווים מוחזק ב- C-LIST, בלוקים בגודל 64 תווים, עם מונה ומצביע לבלוק התווים הבא. המידע לא עובר ישירות מה- C-LIST לתהליך, אלא עובר דרך חלק של הגרעין שנקרא LINE DISCIPLINE שפועל כמו פילטר, לוקח את רצף התווים ממתאם המסוף ומבצע מה שנקרא COOKED CHARACTER STREAM – שזה ניקוי תווים מיותרים, זיהוי תווים מיוחדים וכדומה. לאחר מכן המידע יועבר לתהליך.
במצב של RAW MODE – התהליך מוותר על הסינון של LINE DISCIPLINE.