

Files



© 2022 מערך הפעלה

Operating Systems

Lecture 9 – File Systems

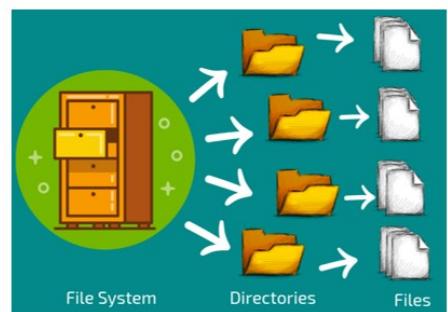
Dr. Marina Kogan-Sadetsky

נושא יחסית קטן: בагודל 2 הרצאות.

נדבר על קבצים מנוקדת המבט של מערכת הפעלה. לרוב אנו נדונם בדיסק קשיח, למחרות שכיוון כל המחשבים עובדים עם SSD כי הוא עדין בשימוש וקל להסביר עליו.

Course Syllabus

1. Introduction
2. Process Management
3. Scheduling algorithms
4. Synchronization
5. Memory Management
6. **File Systems**
 - Concepts
 - File system implementation
 - NTFS
 - NFS
7. Virtualization



Concepts

File Systems : outline

Concepts

File system implementation

- Disk space management
- Reliability
- Performance issues

NTFS

NFS

File Systems

Answers three major needs:

- Large & cheap storage space
- Non-volatile storage that is not erased when the process using it terminates
- Way to share information between processes

מערכות קבצים

אפליקציות מחשב מחייבות אחסון והעברה של מידע, אך מרחב הכתובות של התהיליך מגביל במקרים מסוימים, במיוחד כאשר מדובר באפליקציות שדורשות כמויות מידע אדירות, כמו מערכות להזנת טיסות, בנקים ועוד. בנוסף, אפליקציות אלו דורשות שהמידע לא יאבוט כאשר התהיליך מסתיים, מכיוון שהן דורשות את המידע לטוויה אחר. כמו כן, "תכנו מצבים בהם מספר תהליכיים ירצו גישה לאותו מידע, מה שחייב שהמידע לא יהיה תלוי בתהיליך מסוים".

מערכות קבצים מגיבות על שלושה צרכים עיקריים:
א. מקום אחסון גדול וול

ב. אחסון שאינו מתחמק כאשר התחליק שימושה בו מסתיים
ג. דרך לשף מידע בין תהליכיים

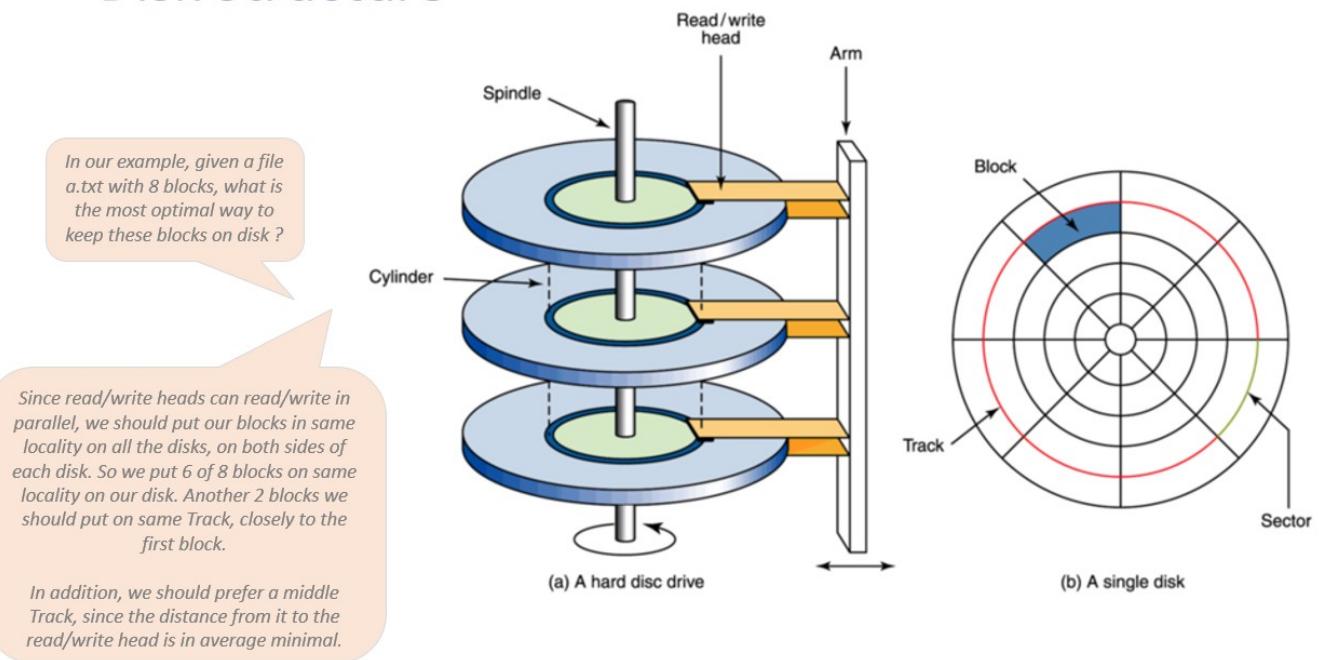
ישן מספר פתרונות אחסון נפוצים לביעות אלו, כולל דיסק קשיח מגנטי, SSD והתקני אחסון ניידים. בעצם, ניתן לראות את הדיסקים כסדרות ליניאריות של בלוקים בגודל קבוע שתומכים בפעולות קריאה וכתיבה, מה שיכל לפטור את בעיית האחסון לטווח ארוך.

אך ישן בעיות רבות הקשורות כאשר משתמשים בדיסק קשיח לאחסון לטווח ארוך, במיוחד במקרים אדולות עם מספר משתמשים. לדוגמה, איתור מידע, הגנה על פרטיות המידע וניהול בלוקים חופשיים.

הקבצים הם יחידות לוגיות של מידע שאין תלויות אחת בשנית ומיעדרות למודל את הדיסק. האחוריות על ניהול הקבצים היא של מערכת הפעלה. ניהול הקבצים מתבצע על ידי מערכות קבצים, שמנהלות קבצים ומערכות עיקריות בעיצוב מערכת הפעלה.

בשלב זה, אנו נדון במבנה הפיזי של הדיסק הקשיח, ולאחר מכן נדון בקבצים ובמערכות קבצים.

Disk Structure



מבנה הדיסק הקשיח

אחסון לא נשכח

נכילים בקצבча זו דיסקים מגנטיים (DISKS) או SSDs, זיכרנו קבוע. הדיסק הקשיח, שהוא הכלי ישן והכי איטי, יכול להיות יותר זול פי 100 מה-RAM ואף יהיה בעל קיבולת גדולה יותר פי 100. הבעיה היא היחידה היא שזמן הגישה האקראי לננתונים עליו הוא איטי במספר בעל 3 ספרות. הסיבה היא שדיסק הוא מכשיר מכני, כמו שנייתן לראות בתמונה.

הdisk הקשיח מורכב מפלטה אחת או יותר שמסתובבות ב-5400, 7200, 10,800, 15,000 RPM או יותר. יש לה זורע מכנית אשר מסתובבת מעל הפלטה מהפינה, בדומה לזרוע ההפקה בפטיפון ישן לניגון תקליטים ויניל. המידע נכתב על הדיסק בסדרת מעגלים מרכזים. בכל מיקום זורע מסוים, כל אחד מהראשים יכול לקרוא אזור עגול הידוע כטרק. יחד, כל הטרקים למיקום זורע מסוים מהווים צילינדר.

כל טרק מחולק למספר מסוים של סקטורים, בדרך כלל 512 בתים לסקטור. בדיסקים מודרניים, הצלינדרים החיצוניים מכילים יותר סקטורים מאשר הפנימיים. העברת הזרוע מצילינדר אחד לאחר מסתיים בערך ב-1 מילישניה. העברתה לצילינדר אקראי לוקחת בדרך כלל 10-5 מילישניות, בהתאם לדיסק. ברגע שהזרוע נמצאת על הטרק הנוכחי, הדיסק חייב לבחوت שהסקטור הנדרש יסתובב מתחת לראש, עיכוב נוסף של 10-5 מילישניות, בהתאם ל-RPM של הדיסק. ברגע שהסקטור נמצא מתחת לראש, הקראיה או הכתיבה מתבצעת בקצב של 50 sec/MB בדיסקים בסיסיים ועד 200-160 MB/sec בדיסקים מהירים יותר.

בדוגמה שלנו, נניח שיש לנו קובץ בשם txt.a עם 8 בלוקים, מה הדרך האופטימלית ביותר לשמר את הבלוקים האלה על הדיסק? לאחר וראשי הקראיה/הכתיבה יכולים לקרוא/לכתוב במקביל, علينا לשים את הבלוקים שלנו באותו המיקום בכל הדיסקים, בשני הצדדים של דיסק. לכן, אנו שמים 6 מתוך 8 הבלוקים באותו המיקום בדיסק שלנו. שני הבלוקים הנוספים علينا לשים באותו הטרק, קרובים לבלוק הראשון.

בנוסף, علينا להעדיף טרק באמצע, לאחר הקראיה/הכתיבה הוא בנקודת המינימלי.

הדיםק הקשי הוא אוסף של דיסקיות שנitinן לכתוב ולקראן מהן מלמטה ומלמעלה. אוסף הדיסקיות האל מסתובב סביב מרכזו באמצעות ציר. יש גם מחרט קראיה וכתיבה שקוראת וכותבת מהדיסקיות. לבסוף, יש את הזרוע שמחזיקה את המחטיים.

כל דיסקית מחולקת לסקטורים, כל סקטור מחולק לטרקים ובין כל שני טרקים יש בלוק. כפי שנitinן לראות, טרקים שרחוקים מהמרכז מהווים בלוק אחד או שמספר טרקים קטנים מהווים בלוק אחד.

File Structure

- **Unstructured**

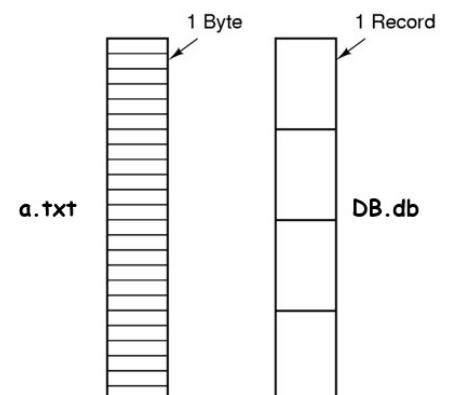
- for OS, unstructured file is just a **sequence of bytes**
- examples: txt file, word file, ppt file, c file, ...

- **Structured**

- OS can parse the file and use it (i.e., run it, manage it)
- examples: executable files (ELF), directory files, IO files

- **Files with records structure**

- file is a sequence of records
- read/write operate is on full record only
- examples: directory files, database files



מבנה קובץ

קובציים יכולים להיות מבנים שונים, כאשר ישנו שלושה אפשרויות נפוצות:

א. **Unstructured**: למערכת ההפעלה, קובץ מסווג זה הוא רק סדרה של בתים ללא משמעות מיוחדת. ככלומר, הקובץ הוא סדרה של בתים ללא מבנה מוגדר מראש, מה שמאפשר מקסימום גמישות. דוגמאות לקובצים מסווג זה הן קבצי c, txt, word, ppt ועוד.

ב. לבתים בקובץ יש משמעות למבנה הפעלה, כלומר מערכת הפעלה יכולה לאזרז את המשמעות של הקבצים ולהשתמש בהם, כמו הריצתם וניהולם. דוגמאות לקבצים מסוג זה הן קבצים הנחוצים להריצה (ELF), קבצי ספריות, קבצי ISO.

ג. **Files with records structure**: הקובץ הוא סדרה של רשומות, כאשר פעולות הקריאה/הכתיבה מתבצעות על רשומה מלאה בלבד. דוגמאות לקבצים מסוג זה הן קבצי ספריות, קבצי מסדי נתונים.

במבנה הראשון, המערכת הפעלה מתייחסת לקבצים כסדרה לא מבונת של בתים, מה שמספק את האמירות המרבית. תוכניות משתמש יכולות לשים כל דבר שהן רוצות בקבצים שלהם ולתת להם שמות בכל דרך שהן מוצאות נוחה. המערכת הפעלה לא מסיעת, אך היא גם לא מפrieveה. עבור משתמשים שרצו לעשוט דברים לא שגרתיים, הדבר האחרון יכול להיות חשוב מאוד. כל הוראות של UNIX (כולל Linux ו-Windows) וגם MacOS משתמשות במודול קובץ זה.

השלב הראשון במבנה הוא קובץ שהוא סדרה של רשומות באורך קבוע, כל אחת עם מבנה פנימי מסוים. מרכזיו לרעון של קובץ הוא סדרה של רשומות הוא הרעיון שפעולות הקריאה מחזירה רשומה אחת ופעולות הכתיבה מחליפה או מוסיף רשומה אחת.

הסוג השלישי של מבנה קובץ מוצג בדוגמה של קובץ שהוא עצם של רשומות, לא בהכרח באותו האורך, כאשר כל אחת מהן מכילה שדה מפתח במיקום קבוע ברשומה. העץ ממוקן לפי שדה המפתח, כדי לאפשר חיפוש מהיר של מפתח מסוים.

File Types

- **Commonly-used files**

- ASCII
- binary

- **System files**

- files which represent directories
- files which represent IO devices
 - character I/O
 - block I/O

*files that represent I/O
device that gets its input in
characters one by one –
keyboard, screen, network
channel*

*files that represent I/O
device that gets its input in
blocks of characters – disk*

סוגי קבצים

מערכות הפעלה תומכות במגוון סוגים של קבצים. לדוגמה, UNIX (כולל MacOS ו-Linux) ו-Windows משתמשים בקבצים רגילים ובתקיות. UNIX גם משתמש בקבצים מיוחדים של תווים ובלוקים. קבצים רגילים הם אלה שמיכלים מידע של המשתמש. תקיות הן קבצי מערכות שימושיים לשם רירה על מבנה מערכת הקבצים.

קובצים רגילים באופן כללי הם או קבצי ASCII או קבצים בינהירים. קבצי ASCII מורכבים משורות של טקסט. במערכות מסוימות, כל שורה מסתירה בתו של חזרה לשורה. באחרות, משתמשים בתו של קפיצה לשורה. מערכות מסוימות (למשל, Windows) משתמשות בשני התווים. השורות לא צריכים להיות באותו האורך.

היתרון הגדול של קבצי ASCII הוא שנית להציג אותם ולהדפיס אותם כפי שהם, ונitin לעורך אותם עם כל עורך טקסט. בנוסף, אם מספר גדול של תוכניות משתמשות בקבצי ASCII לקלט ולפלט, קל לחבר את הפלט של תוכנית אחת לקלט של תוכנית אחרת, כמו בzinorot shell.

קובצים אחרים הם בינהירים, שפשט אומר שהם לא קבצי ASCII. רישימתם על המדפסת תיתן רישימה בלתי מובנת מלאה בזבל אקראי. בדרך כלל, יש להם מבנה פנימי שידוע לתוכניות משתמשות בהם.

File attributes

- **general info**
file name, creator, owner, creation time, last-access time, ...
- **location on disk, size, ...**
- **ASCII/binary flag, system file flag, hidden flag, ...**
- **protection, password, read-only flag, ...**

מאפייני קובץ

כל קובץ יש שם ונתונים. בנוסף, כל מערכות הפעלה מקשרות מידע נוסף עם כל קובץ, לדוגמה, התאריך והשעה שבה הקובץ שונה לאחרונה וגודל הקובץ. נקרא לפריטים הנוספים האלה מאפייני הקובץ. חלק מהאנשיים קוראים להם מטא-דאטה. הרשימה של המאפיינים משתנה מאוד ממערכת למערכת. אין מערכת קיימת שיש לה את כל אלה, אך כל אחד מהם קיים במערכת מסוימת.

ARBUTHE המאפיינים הראשונים קשורים להגנת הקובץ ומספרים מי יכול לגשת אליו ומי לא. ישנן תוכניות שונות אפשריות, חלק מהן נלמד בהמשך. במערכות מסוימות חייב להציג סיסמה כדי לגשת לקובץ, במקרה זה הסיסמה חייבת להיות אחד מהמאפיינים.

הdaglim הם סיבות או שדות קצרים שמשלטים אואפשר מאפיין מסוים. קבצים מסוימים, לדוגמה, לא מופיעים בראשיותם של כל הקבצים. דגל הארכיו הוא סיבית שמקבתת אחרי האם הקובץ גובה לאחרונה. תוכנית הגיבוי מנקה אותה, ומערכת הפעלה מגדרה אותה בכל פעם שקובץ משתנה. בדרך זו, תוכנית הגיבוי יכולה לספר אילו קבצים צריכים להיגבות. הדגל הזמן מאפשר לסמן קובץ למחיקה אוטומטית כאשר התהילך שייצר אותו מסתיים.

שודות אורך הרשימה, מקום המפתח, ואורך המפתח נמצאים רק בקבצים שרשומותיהם יכולות להיחפש באמצעות מפתח. הם מספקים את המידע הנדרש כדי למצוא את המפתחים.

הזמןים מעקבים אחריו מועד יצירת הקובץ, הגישה האחרונה אליו, והשינוי האחרון בו. הם שימושיים למגוון מטרות. לדוגמה, קובץ מקור שונה לאחר יצירת הקובץ האובייקט המתאים צריך להיקmpl מחדש. שודות אלה מספקים את המידע הנדרש.

הגודל הנוכחי מסpter כמה הקובץ גדול ברגע זה. מערכות הפעלה ישנות של מחשבים מרכזיים דרשו שהגודל המרבי יצוין כאשר הקובץ נוצר, כדי לאפשר למערכת הפעלה לשמור מראש את הכמה המרבית של האחסן. מערכות הפעלה של מחשבים אישיים מספיק חכמתה להסתדר ללא תcona זו בימים אלה.

File Operations

- **Create - Delete**

- **Close - Open**

- **Read - Write**

*operations performed on the **current location***

- **Seek** - *a system call to move **current location** to some specified location*

- **Get Attributes**

- **Set Attributes** - *for attributes like name; ownership; protection mode; "last change date"*

פעולות על קבצים

קבצים קיימים כדי לאחסן מידע ולאפשר את שחזורו מאוחר יותר. מערכות שונות מספקות פעולות שונות לאפשר אחסון ומחזור. להלן דיוון על הפעולות למערכת הנפוצות ביותר הקשורות לקבצים.

א. **יצירה:** הקובץ נוצר ללא נתונים. מטרת הקריאה היא להודיע שהקובץ בדרך ולהגידו חלק מהמאפיינים.

ב. **מחיקה:** כאשר הקובץ לא נדרש יותר, יש למחוק אותו כדי לשחרר מקום בדיסק. תמיד יש קריאה למערכת למטרה זו.

ג. **פתיחת:** לפני שימוש בקובץ, על התהילה לפתח אותו. מטרת הקריאה לפתיחת היא לאפשר למערכת להביא את המאפיינים ורשימת כתובות הדיסק לזכרון הראשי לגישה מהירה בקריאות מאוחרות.

ד. **סגירה:** כאשר כל הפעולות הן מושלמות, המאפיינים וכתוות הדיסק לא נדרשות יותר, ולכן יש לסגור את הקובץ כדי לשחרר מקום בטבלה הפנימית. מערכות רבות מעודדות זאת על ידי הטלת מקסימום מספר קבצים פתוחים על תהליכי. דיסק נכתב בבלוקים, וסגירת קובץ מאלצת כתיבה של הבלוק האחרון של הקובץ, אף על פי שהבלוק עשוי לא להיות מלא לפחות עדין.

ה. **קריאה:** נתונים נקראים מהקובץ. בדרך כלל, הבטים מגיעים מהמקום הנוכחי. המתקשר חייב לציין כמה נתונים נדרשים וגם לספק חוץ לשים אותם בו.

ו. **כתיבה:** נתונים נכתבים לקובץ שוב, בדרך כלל במקום הנוכחי. אם המקום הנוכחי הוא סוף הקובץ, גודל הקובץ מתרחב. אם המקום הנוכחי הוא באמצע הקובץ, הנתונים המקוריים מוחלפים ונאבדים לנצח.

ז. **הוספה:** קריאה זו היא צורה מוגבלת של כתיבה. היא יכולה להוסיף נתונים רק לסוף הקובץ. מערכות שמספקות סט מינימלי של קריאות למערכת בדרך כלל אין להן הוספה, אך חלק מהמערכות יש להן קריאה זו.

ח. **חיפוש (seek):** לקבצים באירוע אקריאת, נדרש שיטה לציין מאיפה לחת את הנתונים. גישה נפוצהachahet היא קריאה למערכת, חיפוש, שמצויה מחדש מצביע הקובץ למקום מסוים בקובץ. לאחר שקריאה זו הושלמה, ניתן לקרוא מידע מהמקום הזה, או לכתוב אליו.

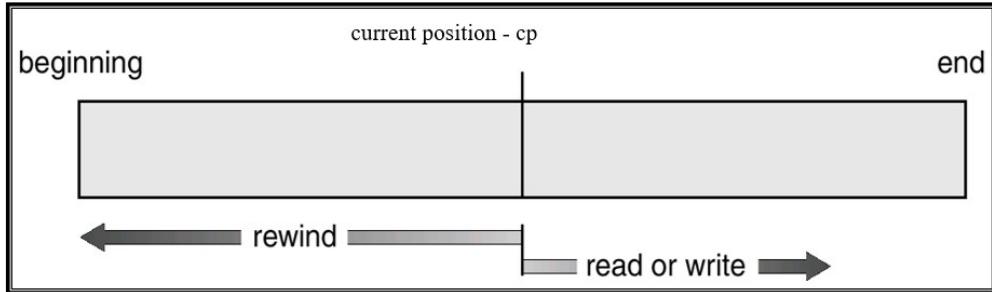
ט. **קבלת מאפיינים:** תהליכי ליעיתם קרובות צרכים לקרוא מאפייני קובץ כדי לבצע את עבודותם. דוגמה, התוכנית make XINU משמשת באופן נרחב לניהול פרויקטים של פיתוח תוכנה המורכבים במספר רב של קבצי מקור. כאשר מפעילים את make, היא בודקת את זמני השינוי של כל הקבצים המקוריים והאובייקט ומסדרה את מספר ההדרות המינימלי הנדרש כדי לעדכן הכל. כדי לבצע את משימתה, היא חייבת להסתכל על המאפיינים, כולל, זמני השינוי.

י. **הגדרת מאפיינים:** חלק מהמאפיינים הם ניתנים להגדירה על ידי המשתמש ונitinן לשנותם לאחר שהקובץ נוצר. קריאה למערכת זו הופכת את זה לאפשרי. מידע מצב ההאגנה הוא דוגמה ברורה. רוב הדגלים נמצאים גם בקטגוריה זו.

יא. **שינוי שם:** קריאה זו אינה חיונית לאחר שניתן להעתק קובץ שצריך להיות מוחלף ואז למחוק את הקובץ המקורי. אך, שינוי שם של סרט בנפח של 50 גיגה-בייט על ידי העתקתו ואז מחיקת המקור ייקח זמן רב.

Sequential-access files

- read file bytes/records sequentially, from the beginning
- cannot jump around, but can rewind (i.e., go back to start position)



Random access files

can go forward / backward on file

- read bytes/records in any order
 - receive a **position-in-file parameter**
 - call for *seek(position-in-file)* to get the required position in file
 - read/write from the current position in file

*all files of modern
operating systems are
random access*

קבצים בעלי גישה סדרתית וrndומית

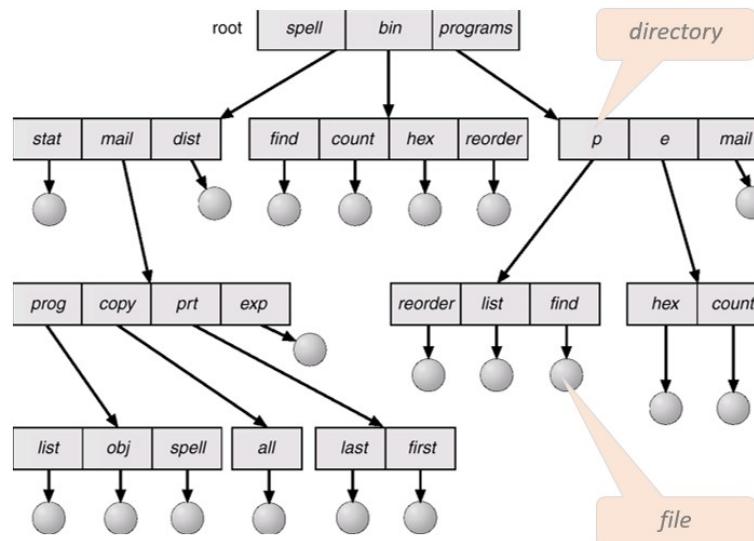
קבצים בעלי גישה סדרתית:

הגישה הסדרתית לקבצים הייתה נהוגה בעבר, אך היום הגישה הרנדומית הופכת להיות נהוגה יותר. בגישה הסדרתית, כדי להגיע לבית מסויים, علينا להתחיל מהבית הראשון ולקרוא באותו סדר עד למגעים לבית הרצוי. ב的日子里 אחרים, אנו קוראים את הבטים או הרשומות של הקובץ בסדר, מהתחלת. איננו יכולים לדלג סתם כר, אך יכולים להקפי אחרה, ככלומר, לחזור למקום ההתחלת.

קבצים בעלי גישה רנדומית:

בגישה הרנדומית, ניתן לקרוא בתים בכל סדר. אנו מקבלים פרמטר של מיקום-בקובץ, ואז מבצעים קריאה לפונקציה `seek` עם המיקום המבוקש בקובץ. זה מאפשר לנו לlect קדימה ואחוריה בקובץ, לקרוא את הבטים או הרשומות בכל סדר שאנו רוצים. אנו קוראים או כתבים מהמיקום הנוכחי בקובץ. כל הקבצים של מערכות הפעלה המודרניות הן בעלות גישה רנדומית.

Tree-Structured Directories



תיקיות

התפקיד של התקיות הוא לארגן ולעקוב אחרי קבצים. ישנן שתי צורות עיקריות של מערכותתיקיות: מערכת TICKIES ברמה אחת ומערכת TICKIES במספר רמות.

מערכת TICKIES ברמה אחת:

במערכת זו ישנה TICKIES אחת שמכילה את כל הקבצים. המערכת הזאת הייתה בשימוש במחשבים האישיים הראשונים ואף בסופר מחשב הראשון. היתרון של מערכת זו הוא הפשטות שלה והיכולת לאתר קבצים במהירות, לאחר שיש רק מקום אחד לחפש בו. עם הזמן, כאשר המידע שהוא אוסףים התרחב, ניהול באמצעות מערכת ברמה אחת הפך ללא פרקטי, והיתה צורך במערכת אחרת.

מערכת TICKIES במספר רמות (או מערכת TICKIES במבנה של עצ':)

מערכת זו מאפשרת ליצור מבנה של TICKIES שדומה לעץ, שבו כל TICKIES יכולה להכיל מספר תחת-TICKIES או קבצים. זה מאפשר למשתמש לארגן את הקבצים שלו בדרךים שונות, כמו כן, מספר משתמשים יכולים להחזיק TICKIES שורש פרטיות בשורת קבצים מסוימת. בדוגמה שלנו, הקודקוד העליון הוא השורש (root), שהוא הסלאש

Directory Operations

- **Create/Delete**
 - file
 - sub-directory
- **Search for a file or sub-directory**
- **List (`ls`) a directory**
- **Rename file in directory**
- **Link a file to a directory (from other directories)**

For directories, the execute permission bit means search permission.

פעולות על תיקיות

ראשית, נדגש כי למערכות הפעלה שונות יש פעולות שונות הקשורות לניהול תיקיות. נתמקד באלה שנפוצות במערכות הפעלה רבות. בהינתן תיקייה, אפשר לפתח אותה לקריאה, למחוק אותה, לחפש אחר קובץ בתיקייה, לשנות את שמה או לקשר קובץ לתיקייה. פעולה נוספת שנראית בהמשך היא `link`.

פעולות נפוצות כוללות:

- א. ייצירה/מחיקה של קובץ או תת-תיקייה.
- ב. חיפוש קובץ או תת-תיקייה.
- ג. רישימה (`ls`) של תיקייה.
- ד. שינוי שם של קובץ בתיקייה.
- ה. קישור של קובץ לתיקייה (מתיקיות אחרות).

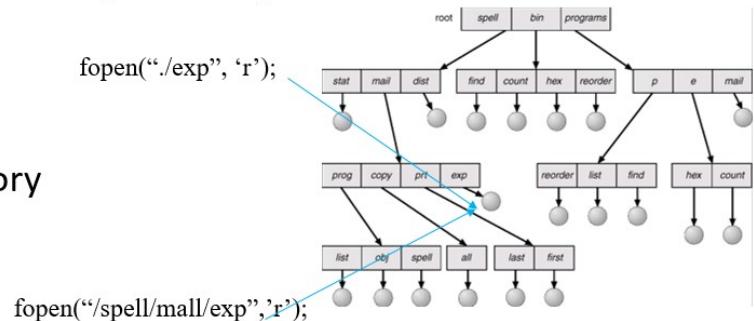
לABI תיקיות, הרשות הביצוע משמעה הרשות חיפוש.

במערכות הפעלה כמו AIX, לדוגמה, ישן פועלות נוספות שנייתן לבצע על תיקיות, כולל ייצירה, מחיקה, פתיחה, סגירה, קריאה, שינוי שם, קישור וביטול קישור. כמו כן, ישן פועלות לניהול מידע ההגנה המשויך לתיקייה.

גורסה משופרת של פעולה הקישור היא הקישור הסמלי (שמכונה לעיתים קישור דרך או כינוי). במקום שניי שמות מצביים לאוטו מבנה נתונים פנימי שמייצג קובץ, ניתן ליצור שם שמצוין לקובץ קטן שמצוין קובץ אחר. כאשר הקובץ הראשון משמש, לדוגמה, נפתח, מערכת הקבצים מעקבת אחר הנתיב ומוצאת את השם בסוף. אז היא מתחילה את ההליך החיפוש מחדש באמצעות השם החדש. קישורים סמליים יש להם את היתרון שהם יכולים להציג אבולות דיסק ואף לצין קבצים במחשבים מרוחקים. המימוש שלהם הוא פחות יעיל מאשר קישורים

Path names

- **absolute path**
 - name starts from root directory
- **relative path**
 - name starts from *the working directory (current directory)*
- each process has its own working directory
 - **shared by threads**
- dot (.) – current directory
 - **cp ./a.txt ./b.txt**
- dotdot (..) – parent directory
 - **cp ../lib/directory/ .**



שמות נתיבים

במערכת תיקיות המאורגנת כעוז, נדרשת דרך לציין שמות של קבצים. שני שיטות נפוצות משמשות למטרה זו.

השיטה הראשונה היא שם הנתיב המוחלט, שמורכב מהנתיב מהתיקייה השורש לקובץ. לדוגמה, הנתיב /usr/ast/mailbox/asm שמעו שבתיקייה השורש יש תת-תיקייה בשם asm, שבה יש תת-תיקייה בשם ast, שבה נמצא הקובץ mailbox.asm. שמות הנתיב המוחלטים תמיד מתחילה בתיקייה השורש והם ייחודיים.

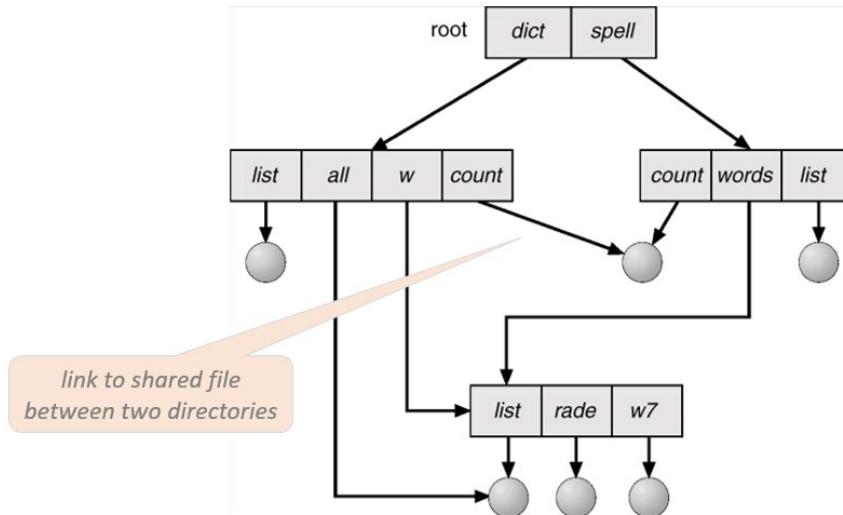
השיטה השנייה היא שם הנתיב היחסית. שימוש בשיטה זו מתרבע בשילוב עם המושג של התיקייה העובדת (או התיקייה הנוכחיית). משתמש יכול לקבוע תיקייה אחת כתיקייה העובדת, ואז כל שמות הנתיב שאינם מתחילה בתיקייה השורש מתקבלים ביחס לתיקייה העובדת. לדוגמה, אם התיקייה העובדת הנוכחיית היא /usr/hjb/, אז הקובץ שם הנתיב המוחלט שלו הוא /hjb/mailbox/mailbox.asv יכול להתייחס אליו פשוט כמו asv.

כל תהליך יש את התיקייה שלו שהוא עוקב עליה (שימושפת לכל הת'רידים). נקודה מתייחסת לתיקייה הנוכחיית, ונקודותים מתייחסות לתיקיית האב.

בדוגמאות, המערכת מבינה מה הנקודה בנתיבים מכיוון שיש ב-PCB מצביע לקובץ שהוא עובד עליו.

Directed-Acyclic-Graph (DAG) Directories

- Allows sharing directories and files



תיקיות בצורת גרף מכוון ללא מעגלים (DAG)

אפשרות זו מאפשרת שיתוף של תיקיות וקבצים. ניתן ליצור קישור לקובץ משותף בין שתי תיקיות. הרעיון הוא שישנם שני עליים שמצבעים לאותו הקובץ. האIOR הוא אגרי לוחוטין ומשמש להמחשת הרעיון, ולא להסתמך עליו. נחזר לנושא זה מאוחר יותר.

Locking files

- any part of a file may be locked, to prevent race conditions
- locks are **shared** (for reading) or **exclusive** (for writing)
- blocking or non-blocking possible (blocked processes awakened by OS)
- lock is removed when file is closed, or when process terminates
- Unix provides **advisory** file locking

Advisory locking means that processes should cooperate "peacefully" without OS being involved in lock management. Indeed OS lets process to use files without asking for a lock.

flock(file descriptor, shared / exclusive)

flock () is basic lock function

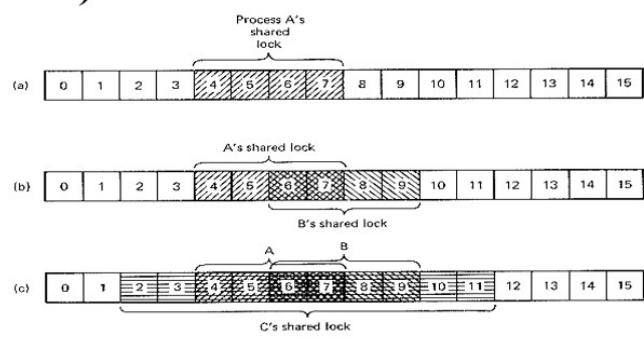


Fig. 7-11. (a) A file with one lock. (b) Addition of a second lock. (c) A third lock.

נעילת קבצים

נעילת קבצים (ביוניקס) מאפשרת בקרת גישה על חלקים בקובץ על מנת לאפשר מניעה הדדית על הגישה לקובץ. מנועלים אלו יכולים להיות משותפים, כלומר אפשריים במספר תהליכיים לקרוא מהקובץ, או במצב של בלדיות.

כלומר לתהיליך אחד בלבד מוענקת היכולת לכתוב לקובץ כל פעם. בנוסף למתקנת יש אפשרות לבחור בין מניעולים אשר חסמים תהליכי עד שיקבלו את המפתח או מניעולים שאינם חסמים תהליכים, כאשר מערכת הפעלה מעירה את התהיליך אם המפתח לא זמין מידית.

כאשר קובץ נסגר או כאשר התהיליך שיצר את המניעול סיים את ריצתו, המניעול נמחק אוטומטית. בנוסף, יוניקס מחלילה את מדיניות advisory file locking. לפי מדיניות זו, מצופה מהתהליכים לשתף פעולה, ללא מעורבות ישירה מצד מערכת הפעלה. במקרים אחרים, מערכת התהליכים להשתמש בקבצים מבלתי דרושים בצורה מפורשת מפותחות למניעולים.

ב尤ニיקס, הפונקציה העיקרית למנגנון זה הוא פונקציית `flock()`:

```
#include <sys/file.h>

int flock(int fd, int operation);
```

הารוגומנט הראשון הוא ה-`fd` של הקובץ פתוח והארוגומנט השני מציין את הפעולה המבוקשת מהקובץ. הארוגומנט השני יכול לקבל אחד מבין הערכים הבאים:

`LOCK_SH`

הפעלת מניעול משותף, דבר המאפשר למספר תהליכים לקרוא את תוכן הקובץ בו זמן נתון.

`LOCK_EX`

הפעלת מניעול בלבד, דבר המאפשר לתהיליך אחד בלבד לכתוב לקובץ בכל פעם.

`LOCK_UN`

הסרת מניעול קיים המוחזק על ידי התהיליך אשר ביצע את הקריאה.

על מנת לאפשר בקשנות שאין חסמת את התהיליך, אפשר להוסיף את `LOCK_NB` לפעולה המבוקשת בשילוב עם הפעולה הבינארית `OR`.

הערות נוספת:

א. המניעולים קשורים ל-`fd` של קבצים פתוחים, דבר המאפשר למספר `fd`-ים שונים להתאחד באותו מניעול. מכאן שגם אם תהיליך השיג מספר `fd`-ים לאותו הקובץ, הם מנוהלים בצורה עצמאית על ידי הקריאה, ככלומר הקריאה יכולה לדוחות ניסיונות לניעול קובץ דרך `fd`-ים שונים אם יש כבר מניעול פעיל על הקובץ.

ב. המניעולים נשמרים גם לאחר ביצוע קריאות מערכת משפחתיות - `exec`.

File Systems : outline

❑ Concepts

❑ File system implementation

- Disk space management

- Reliability

- Performance issues

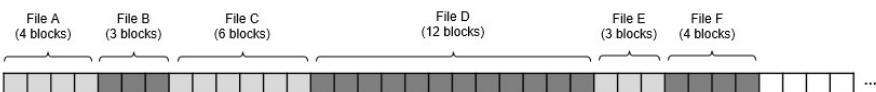
❑ NTFS

❑ NFS

Contiguously allocated files

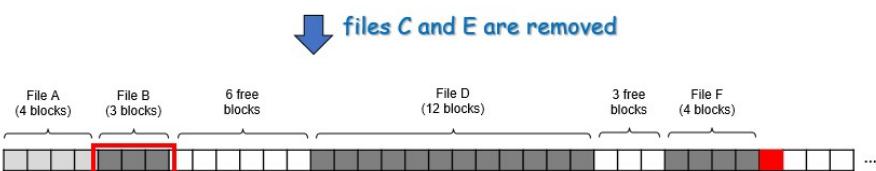
good choice for read-only data

- fast access
- external fragmentation
- file size is not known in ahead – how many space to allocate to file ??



since disk is not in intensive usage all the time, disk compaction is an acceptable solution

no need to move
read/write head when
read or write file



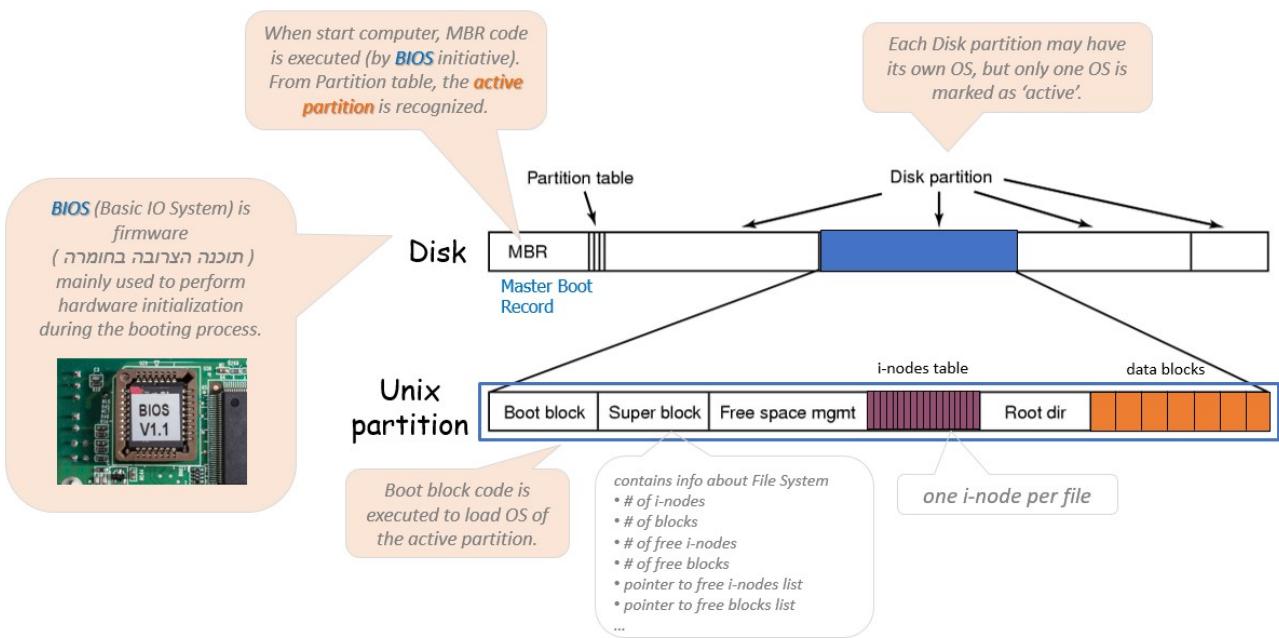
if file needs additional
block(s), it might be no more
contiguously allocated

external
fragmentation

external
fragmentation

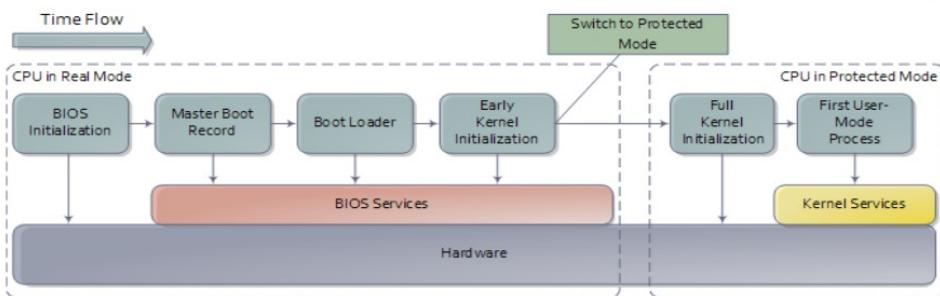
File B
additional block

Unix Disk Partition Structure



Starting an Operating System(Booting)

Linux Booting Process Steps - RHEL 8
<https://www.youtube.com/watch?v=W51A8Ik26Xg>
 Linux Boot Process
<https://www.youtube.com/watch?v=ZtVpz5VWjAs>



- ✓ Power On Switch sends electricity to the motherboard on a wire called the **Voltage Good line**.
- ✓ If the power supply is good, then the **BIOS (Basic Input/Output System) chip** takes over.
- ✓ In Real Mode, CPU is only capable of using approximately 1 MB of memory built into the motherboard.
- ✓ The BIOS will do a **Power-On Self Test (POST)** to make sure that all hardware are working.
- ✓ BIOS will then look for a small sector at the very beginning of your primary hard disk called **MBR**.
- ✓ The MBR contains a list, or map, of all of the **partitions** on your computer's hard disk (or disks).
- ✓ After the MBR is found the **Bootstrap Loader** follows basic instructions for starting up the rest of the computer, including the operating system.
- ✓ In Early Kernel Initialization stage, a smaller core of the Kernel is activated.
- ✓ This core includes the **device drivers** needed to use computer's **RAM chips**.

מימוש מערכת קבצים

כעת, נעבור מנקודת המבט של המשתמש לנקודת המבט של מתקנת המערכת הקבצים. בעוד שהמשתמשים מתמקדים בשמות הקבצים, בפעולות שניתן לבצע עליהם, במבנה התיキות ובונושים אחרים שקשורים לממשק שמערכת הקבצים מספקת, מתקנת מערכת הקבצים מתמקדים באחסון הקבצים והתיキות, בניהול שטח הדיסק ובוודאי שהמערכת פועלת באופן אמין ויעיל.

אנו נדון במגוון תחומיים קשוריים למימוש מערכת קבצים, נטפל בנושאים רלוונטיים, בדברים שצרכיר לחתות בחשבון ובסיקולי תמורה.

ניהול שטח הדיסק

הערה: חלק זה מתיאחס לניהול שטח הדיסק בגישה ה-MBR. מדובר בגישה ישנה מאוד ולכן הוסףתי בסוף הסיכון הסבר מפורט על הגישה הרלוונטיית יותר - UEFI.

מבנה החלוקה של הדיסק ב-**א-זען** הוא חלק מהמערכת הבסיסית לקלט/פלט (BIOS), שהיא תוכנה שמשמשת בעיקר לאתחול החומרה במהלך תהליך האתחול.

כאשר מחשב מתחילה לפעול, הקוד של ה-MBR מצוי לפועלה (ביווזמת BIOS). מטבלת המחיצות, המחיצה הפעילה נזונה. לכל מחיצת דיסק יכולה להיות מערכת הפעלה משלה, אך רק מערכת הפעלה אחת מסומנת כפעילה. קוד ה-**boot block** מצוי לפועלה כדי לטען את מערכת ההפעלה של המחיצה הפעילה. ה-**superblock** מכיל מידע על מערכת הקבצים, כמו מספר ה-nodes-i, מספר הבלוקים, מספר ה-nodes-i החופשיים, מספר הבלוקים החופשיים, מצביע לרשימת הבלוקים החופשיים ועוד.

ה-nodes-i הם מבנה של מערכת הקבצים שמייצג קובץ אחד.

מתווה מערכת הקבצים

דיסקים בדרך כלל מחולקים למחיצות, כאשר לכל מחיצה יש מערכת קבצים נפרדת שאינה תליה במחיצות אחרות. אנו נדון בחולקה של Master Boot Record. החלוקת היא שונה והוא כבר לא בשימוש, אך הוסףתי בסוף נספח רלוונטי ל-UEFI, אשר רלוונטי לקריאה נוספת או כאשר צוות הקורס יחליט לעבור ל-UEFI.

ה-MBR, אשר נמצא בסקטור 0 של הדיסק, נועד לאתחול המחשב ומכליל את טבלת המחיצות אשר מציניות את הכתובות ההתחלתיות והסופיות של כל מחיצה. כאשר המחשב מאותחל, תוכנית ה-MBR מעתה מוחיץ פעילות, קוראת את בלוק ה-**boot** שלו ומריצה אותו על מנת להעלות את מערכת ההפעלה.

המתווה של מחיצת דיסק משתנה ממערכת קבצים אחרת, אך באופן כללי, היא מכילה מספר רכיבים. הרכיב הראשון הוא ה-**superblock**, אשר מכיל פרמטרים הכרחיים על מערכת הקבצים והוא נתון לנזכרן במהלך פעולה האתחול או אשר יש גישה למערכת הקבצים. הוא בדרך כלל יכול מידע כמו סיווג של מערכת הקבצים, מספר הבלוקים ופרטים מנהליים נוספים.

רכיבים נוספים עשויים לכלול מידע על בלוקים חופשיים במערכת הקבצים, כמו **bitmap** (ראו בהמשך) או רשימה של מצביעים.

ביוניקס:

מערכת ההפעלה מקצת לכל קובץ node-i, שהוא מבנה הנתוניים שמייצג קובץ ומכליל מידע על הקובץ. כמו כן היא מקצת טבלת **inodes** - מה שמאפשר לאחר המידע על בלוקים חופשיים במערכת הקבצים.

תיקיות השורש, אשר מייצגת את ראש מבנה העץ של מערכת הקבצים, עשוי להגיע לאחר מכן.

לא הכל לטובת הקבצים - יש **data blocks** שמיועדים להיות **swapping area**. (שוב מדובר ב-**MBR**, מי שכבר משתמש בזה...)

החלק האחרון של הדיסק מכיל את שאר הקבצים והתיקיות במערכת הקבצים.

מימוש קבצים

נעביר כעת לדין על מימוש מערכת קבצים, ונחקרו שיטות שונות בהן מערכת הפעלה מנהלת את הבלוקים של הדיסק **ששייכים** לקבצים.

הकצתה רציפה

המנגנון הפשטוני ביותר להקצתה קבצים הוא ההקצתה הרציפה. לפי מנגנון זה, כל קובץ מאוחסן כסדרה רציפה של בלוקים בדיסק. לדוגמה, קובץ בגודל של 50 קילובייט יוקצה 50 בלוקים רציפים בדיסק שבו גודל הבלוק הוא 1 קילובייט. אם גודל הבלוק הוא 2 קילובייט, הקובץ יוקצה 25 בלוקים רציפים.

היתרונות של ההקצתה הרציפה הם בפשטות המימוש וביצועי הקריאה. לאחר וניתן לקרוא את כל הקובץ בפעולה אחת, ולא נדרש תנוודה של ראש הקריאה/כתיבה במהלך הקריאה או הכתיבה של הקובץ. עם זאת, ישנו חסרוןמשמעותי: פרגמננטציה חיונית בדיסק.

כאשר קבצים מסוימים מהדיסק, נוצרים "חורים" בדיסק, ועם הזמן הדיסק הופך להיות מקוטע עם חורים וקבצים שאינם השתמשו ברחבי הדיסק. כדי להשתמש מחדש בשטח שהתפנה, ניתן לשמור רשימה של החורים. אך כאשר יש ליצור קובץ חדש, יש צורך לדעת את הגודל הסופי של הקובץ כדי לבחור חור שהוא מספיק גדול לאכלוס אותו.

אם קובץ זוקק לבlokים נוספים, יתכן שהוא לא יהיה רציף יותר, וזה יגרום לפרגמננטציה חיונית. זהו חיסרון שמתגמד ביחס ליתרונו. הבעיה האמיתית בדיסקים מסוג SSD היא שהם מתשחקים במהלך הזמן.

ההקצתה הרציפה היא תוכנה רצiosa אך לא בהכרח תתקיים. השימוש העיקרי בדיסקים כמו CDs-ROMs ו-DVDs

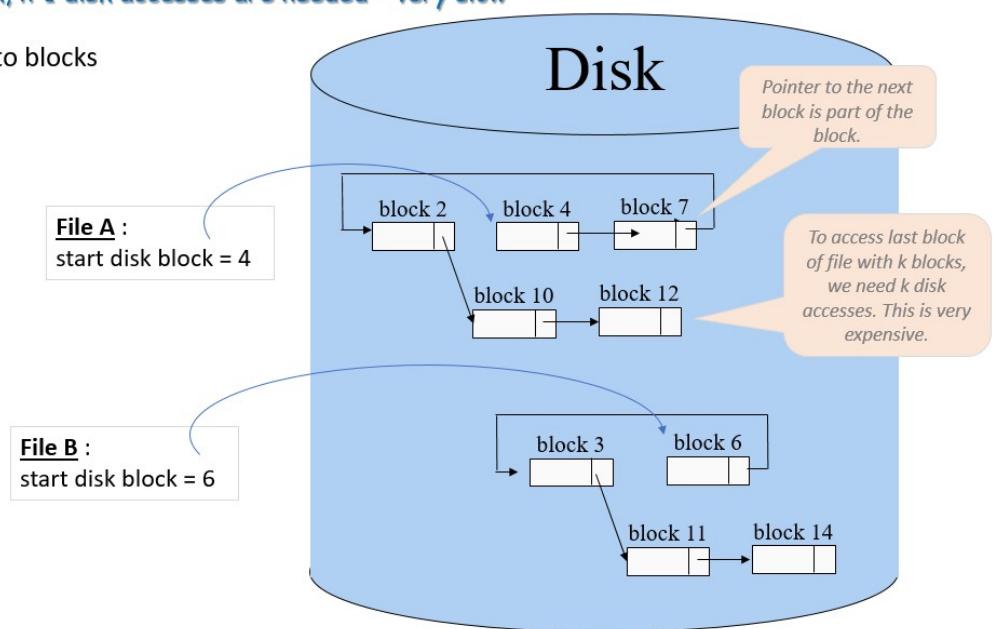
cutet נעביר למימושים עצם של מערכות קבצים. חשוב לציין כי במימוש מערכות קבצים יש מבנה המציג קובץ, מבנה המציג תקיה ומבנה הנתונים בהם היה שימוש במערכת הקבצים. מבנה הקובץ מסגיר את הסביבות של קריאה וכתיבה של בלוקים בדיסק ואיזה מידע נשמר במבנה הקובץ. מבנה התקיה מסגיר את מרכיבות איתור וחיפוש הקבצים.

File as Linked list of disk blocks

to access n'th file block, n-1 disk accesses are needed - very slow

- sequential access to blocks
- weird block size

Since a part of block keeps info about the next block, block size is no more power of 2. This is important for programs that read/write blocks of information and not bytes.

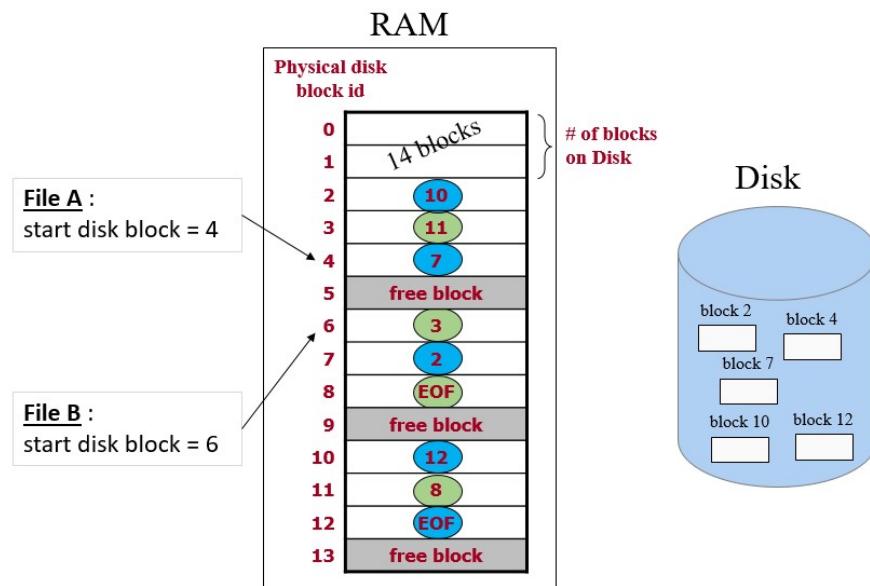


File Allocation Table (FAT) - MSDOS, WINDOWS 95, WINDOWS 98, ...

in-memory all-processes table that stores pointers to all blocks of files

since FAT is in RAM, we need only one disk access to bring the needed block of the file

FAT has entry for each Disk block. For a 1Terabyte Disk, and 4Kb block size, FAT has $2^{40} / 2^{16} = 2^{24} = 16 M$ entries. Each entry of size 40 bits ≈ 5 bytes (to represent block #), so the FAT size is $16 M * 5 \text{ bytes} = 80 Mb$.



הказאת קבצים באמצעות רשימה מקושרת של בלוקים

בשיטה זו, כל קובץ מיוצג כרשימה מקושרת של בלוקים בדיסק, כאשר חלק מהבלוק משמש כמצבייע לבlok הבא, והחלק הנותר משמש לאחסון המידע.

יתרונות:

- ניתול יעיל של שטח האחסון: כל בלוק בדיסק יכול להיות בשימוש,(Cluster), לא נזירת פרוגמנטיה בדיסק.
- ניהילה פשוטה של מידע: כל צורך לשמר הוא את הכתובת של הבלוק הראשון בדיסק, וממנה ניתן לאתר את שאר הבלוקים.

חסרונות:

- א. גישה אקראית איטית: כדי להגיע לבLOCK מס' ח, מערכת הפעלה צריכה לקרוא את 1-ח הבלוקים שלפניו בזורה סדרתית, דבר שמשפיע באופן משמעותי על הביצועים.
- ב. גודל האחסון בבלוק אינו חזקה של 2: לאחר וחילק מהבלוק משמש כמצבי לבLOCK הבא, גודל הבלוק הופך להיות מזר ופחותiesel, לאחר ותוכניות רבות מבצעות פועלות קרייה וכטיבה בבלוקים שגדלים הוא חזקה של 2. בנוסף, קרייה של בלוק שלם דורשת הבאת מידע משני בלוקים ורשורים, מה שמוסיף עומס על המערכת.

ה Katzat קבצים באמצעות טבלת הקצת קבצים (FAT)

בשיטה זו, כל המצביעים לבלוקים של קובץ מאוחסנים בטבלה בזיכרון הראשי, המכונה Table of Contents (FAT). השיטה זו נמשכה במערכות הפעלה Windows 95, MS-DOS, Windows 98, ו-Windows.

“צג קובץ”:

מיוצג על ידי מצביע למיקום בטבלת FAT.

“צג תיקייה”:

התיקייה עצמה מייצגת טבלת הקבצים אשר נמצאים בתיקייה. כל רשומה מחייבת מספר פרטיים אך לצורך איתור קבצים מספיק לשמור את שם הקובץ ומצביע לכתחובת של ה-cluster הראשון של הקובץ.

יתרונות:

- א. גישה אקראית גבוהה: לאחר וכל המצביעים לבלוקים של קובץ נמצאים בזיכרון, ניתן לגשת לכל בלוק בקובץ באופן ישיר, מה שמחיתת את הצורך בגישות לדיסק.
- ב. ניצול מלא של הבלוקים: לאחר והצביעים לא מאוחסנים בתוך הבלוקים עצמם, כל הבלוק יכול לשמש לאחסון מידע.

חסרונות:

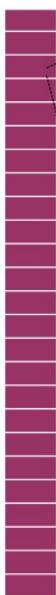
- א. צורך בזיכרון רב: טבלת FAT חייבת להיות בזיכרון כל הזמן, מה שמצריך כמות אדירה של זיכרון, במיוחד עבור דיסקים גדולים. לדוגמה, עבור דיסק בגודל של 1TB עם גודל בלוק של 1KB, טבלת FAT תצריך להכיל כ-1 מיליארד רשומות, כאשר כל רשומה צריכה להיות בגודל של לפחות 3 בתים. לכן, גודל טבלת FAT יהיה בין 3GB ל-2.4GB, תלוי באם המערכת מותאמת למאהירות או לחישכון בזיכרון. זה הופך את השיטה ללא פרקטית עבור דיסקים גדולים.

למרות החסרונות, השיטה זו הייתה בשימוש במערכות הפעלה DOS-MS ובראשות המוקדמות של Windows. בנוסף, השיטה ממשיכה להיות בשימוש במחשבים ניידים ומחשבים משובצים אחרים, כמו מצלמות דיגיטליות, מסגרות תמונות אלקטרוניות, וגני מזיקה, ועוד.

Unix: i-node structure

assume 4Kb disk block size

i-nodes table



i-node of a.txt

File Attributes	
Address of disk block 1	
Address of disk block 2	
Address of disk block 3	
Address of disk block 4	
Address of disk block 5	
Address of disk block 6	
Address of disk block 7	
Address of disk block 8	
Address of disk block 9	
Address of disk block 10	
single indirect	
double indirect	
triple indirect	

mode word
owner
timestamps
size
disk blocks
of links
flags

a.txt

From Wikipedia.

The i-node (index node) is a data structure in a Unix-style file system that describes a file-system object such as a file or a directory. Each node stores the attributes and disk block locations of the object's data.^[1] File-system object attributes may include metadata (times of last change,^[2] access modification), as well as owner and permission data.^[3]

A directory is a list of i-nodes with their associated files. The list includes an entry for itself, its parent, and each of its children. A file system relies on data structures about the files, as opposed to the contents of that file. The former are called metadata—data that describes data. Each file is associated with an i-node, which is identified by an integer, often referred to as an i-number or inode number.

i-nodes store information about files and directories (folders), such as file ownership, access mode (read, write, execute permissions), and file type. On many older file system implementations, the maximum number of i-nodes is fixed at file system creation, limiting the maximum number of files the file system can hold. A special allocation heuristic for i-nodes in a file system is one node for every 2K bytes contained in the filesystem.^[8]

The i-node number indexes a table of nodes in a known location on the device. From the i-node number, the kernel's file system driver can find the file's data blocks. An i-node's file number can be found using the ls -l command. The ls -l command prints the i-node number in the first column of the report.

Some Unix-style file systems such as ZFS, OpenZFS, ReiserFS, btrfs, and APFS use a fixed-size i-node table, but must store equivalent data in order to provide equivalent capabilities.

The operating system kernel's in-memory representation of this data is called struct inode in Linux. Systems derived from BSD fix the size table to include B-trees and the derived B+ trees.

i-nodes do not contain its hard link names, only other file metadata.

Unix directories are lists of association structures, each of which contains one filename and one i-node number.

The file system driver must search a directory looking for a particular filename and then convert the filename to the correct corresponding i-node number.

The operating system kernel's in-memory representation of this data is called struct ino in Linux. Systems derived from BSD fix the name and directory implications.

i-nodes do not contain its hard link names, only other file metadata.

Unix directories are lists of association structures, each of which contains one filename and one i-node number.

The file system driver must search a directory looking for a particular filename and then convert the filename to the correct corresponding i-node number.

The operating system kernel's in-memory representation of this data is called struct ino in Linux. Systems derived from BSD fix the name and directory implications.

Within a POSIX system, a file has the following attributes:^[10]

which may be retrieved by the stat system call.

POSIX i-node description

The POSIX standard mandates file-system behavior that is strongly influenced by traditional UNIX file systems. An i-node is denoted by the phrase "file serial number", defined as a per-file system unique identifier for a file.^[9] That file serial number, together with the device ID of the device containing the file, uniquely identify the file within the whole system.^[10]

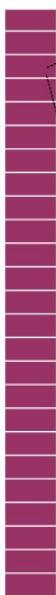
Within a POSIX system, a file has the following attributes:^[10]

which may be retrieved by the stat system call.

Unix: i-node structure

assume 4Kb disk block size

i-nodes table



i-node of a.txt

File Attributes	
Address of disk block 1	data
Address of disk block 2	data
Address of disk block 3	data
Address of disk block 4	data
Address of disk block 5	data
Address of disk block 6	data
Address of disk block 7	data
Address of disk block 8	data
Address of disk block 9	data
Address of disk block 10	data
single indirect	
double indirect	
triple indirect	

i-node entry contains addresses of first 10 data blocks

file may keep 10 blocks * 4 Kb block size = 40 Kb of data

...

divide the file into data blocks, 4 Kb each

a.txt

From Wikipedia.

The i-node (index node) is a data structure in a Unix-style file system that describes a file-system object such as a file or a directory. Each node stores the attributes and disk block locations of the object's data.^[1] File-system object attributes may include metadata (times of last change,^[2] access modification), as well as owner and permission data.^[3]

A directory is a list of i-nodes with their assigned i-node numbers. The list includes an entry for itself, its parent, and each of its children. A file system relies on data structures about the files, as opposed to the contents of that file. The former are called metadata—data that describes data. Each file is associated with an i-node, which is identified by an integer, often referred to as an i-number or inode number.

i-nodes store information about files and directories (folders), such as file ownership, access mode (read, write, execute permissions), and file type. On many older file system implementations, the maximum number of i-nodes is fixed at file system creation, limiting the maximum number of files the file system can hold. A special allocation heuristic for i-nodes in a file system is one node for every 2K bytes contained in the filesystem.^[8]

The i-node number indexes a table of nodes in a known location on the device. From the i-node number, the kernel's file system driver can find the file's data blocks. An i-node's file number can be found using the ls -l command. The ls -l command prints the i-node number in the first column of the report.

Some Unix-style file systems such as ZFS, OpenZFS, ReiserFS, btrfs, and APFS use a fixed-size i-node table, but must store equivalent data in order to provide equivalent capabilities.

The operating system kernel's in-memory representation of this data is called struct inode in Linux. Systems derived from BSD fix the size table to include B-trees and the derived B+ trees.

i-nodes do not contain its hard link names, only other file metadata.

Unix directories are lists of association structures, each of which contains one filename and one i-node number.

The file system driver must search a directory looking for a particular filename and then convert the filename to the correct corresponding i-node number.

The operating system kernel's in-memory representation of this data is called struct ino in Linux. Systems derived from BSD fix the name and directory implications.

Within a POSIX system, a file has the following attributes:^[10]

which may be retrieved by the stat system call.

POSIX i-node description

The POSIX standard mandates file-system behavior that is strongly influenced by traditional UNIX file systems. An i-node is denoted by the phrase "file serial number", defined as a per-file system unique identifier for a file.^[9] That file serial number, together with the device ID of the device containing the file, uniquely identify the file within the whole system.^[10]

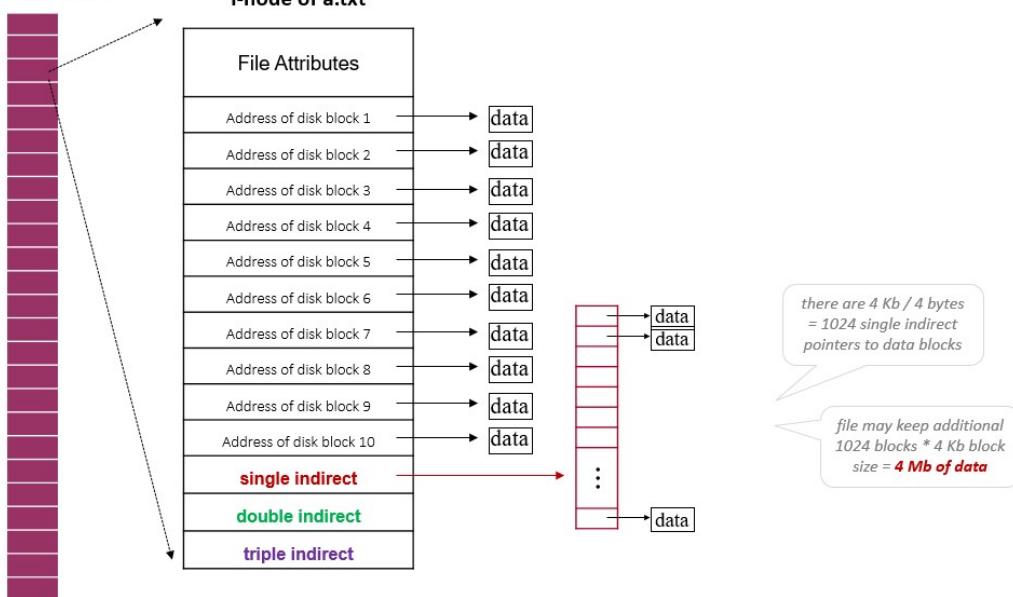
Within a POSIX system, a file has the following attributes:^[10]

which may be retrieved by the stat system call.

Unix: i-node structure

assume 4Kb disk block size

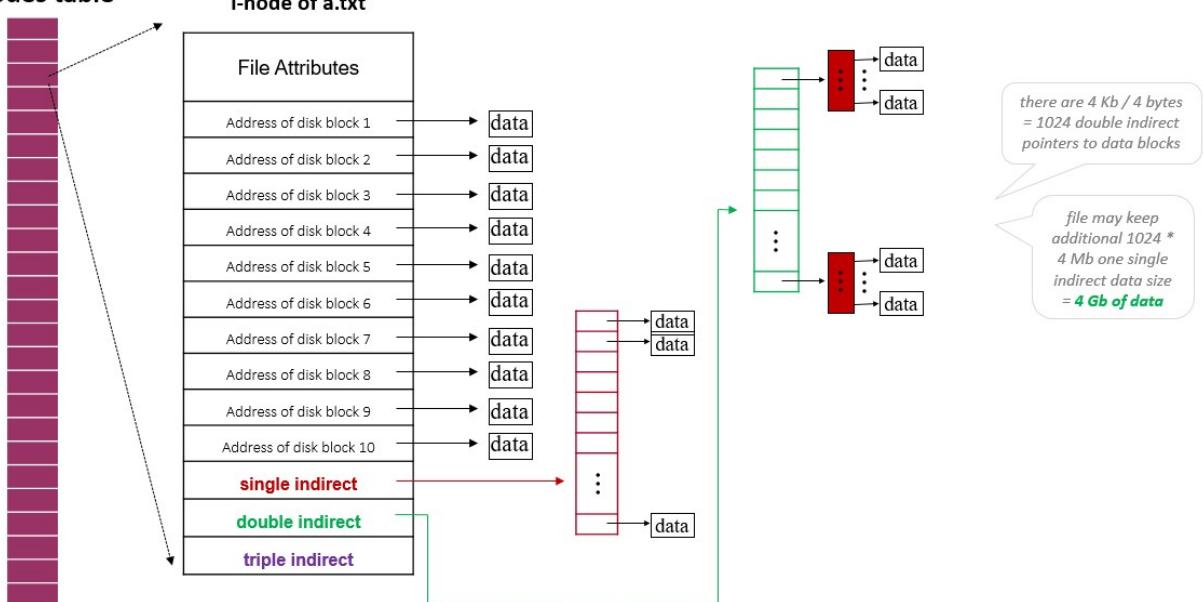
i-nodes table



Unix: i-node structure

assume 4Kb disk block size

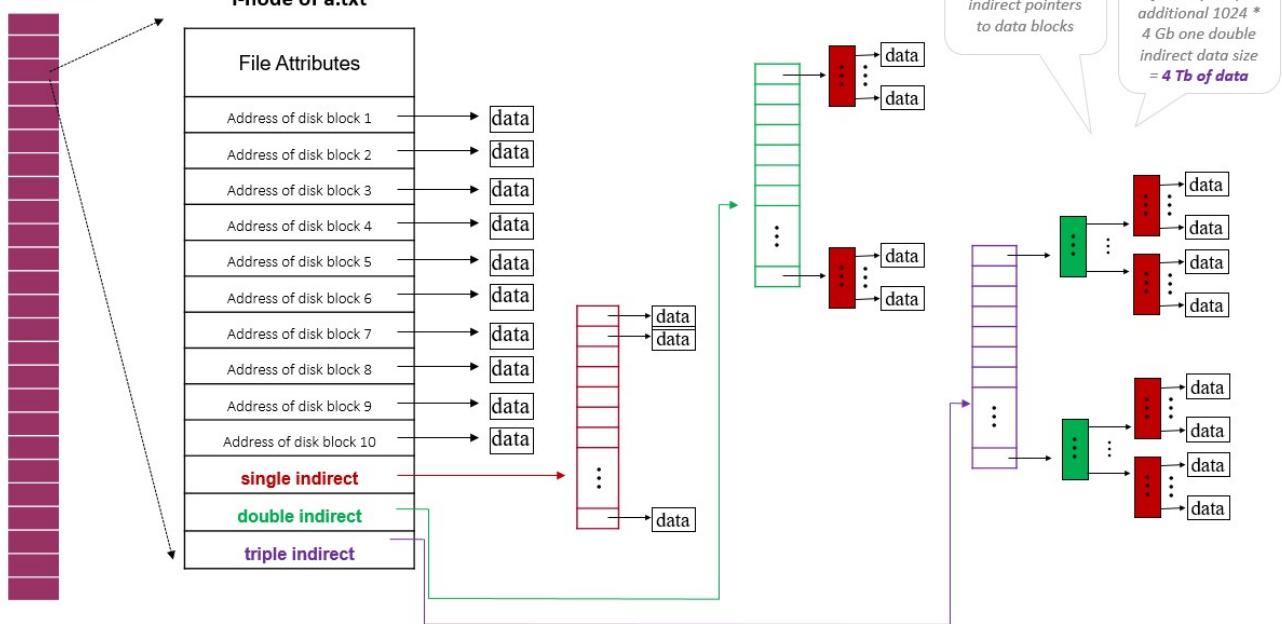
i-nodes table



Unix: i-node structure

assume 4Kb disk block size

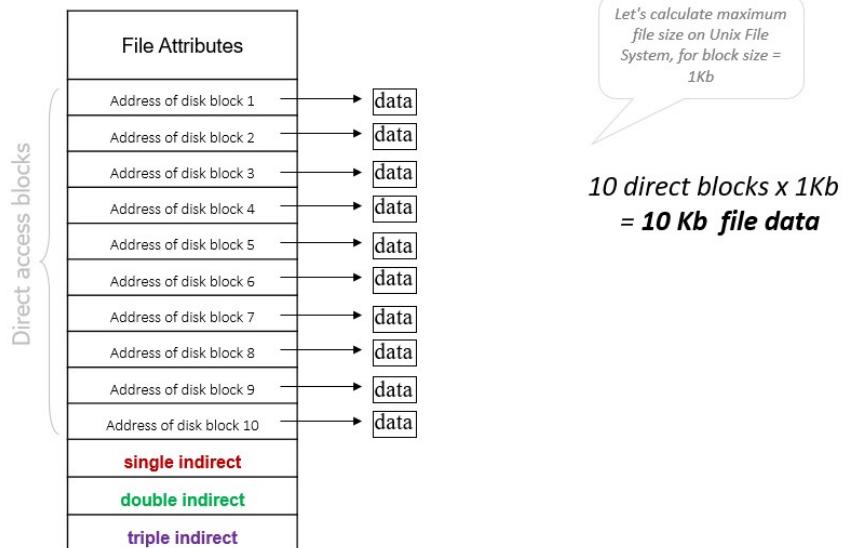
i-nodes table



Unix: i-node structure

assume 1Kb disk block size

i-node of a.txt



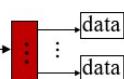
Unix: i-node structure

assume 1Kb disk block size

i-node of a.txt

File Attributes
Address of disk block 1
Address of disk block 2
Address of disk block 3
Address of disk block 4
Address of disk block 5
Address of disk block 6
Address of disk block 7
Address of disk block 8
Address of disk block 9
Address of disk block 10
single indirect
double indirect
triple indirect

*1Kb single indirect block / 4-byte block pointer size
= 256 single direction blocks of 1 Kb
= additional 256 Kb file data*



Unix: i-node structure

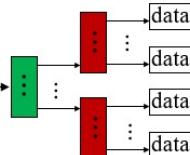
assume 1Kb disk block size

i-node of a.txt

Direct access blocks

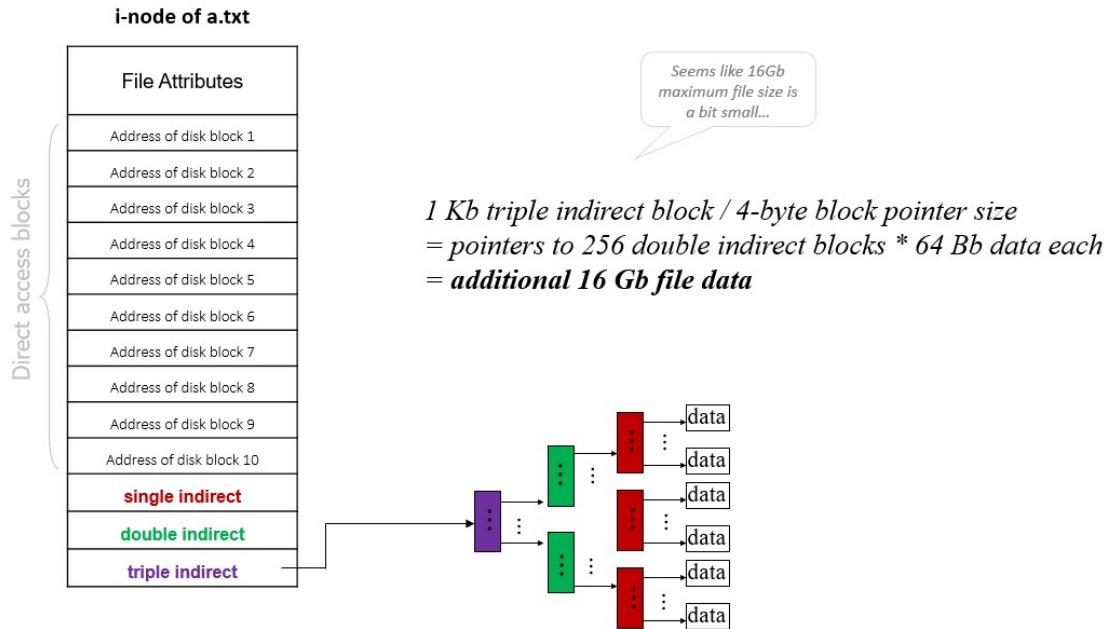
File Attributes
Address of disk block 1
Address of disk block 2
Address of disk block 3
Address of disk block 4
Address of disk block 5
Address of disk block 6
Address of disk block 7
Address of disk block 8
Address of disk block 9
Address of disk block 10
single indirect
double indirect
triple indirect

*1 Kb double indirect block / 4-byte block pointer size
= pointers to 256 single indirect blocks * 256 Kb data each
= additional 64 Mb file data*



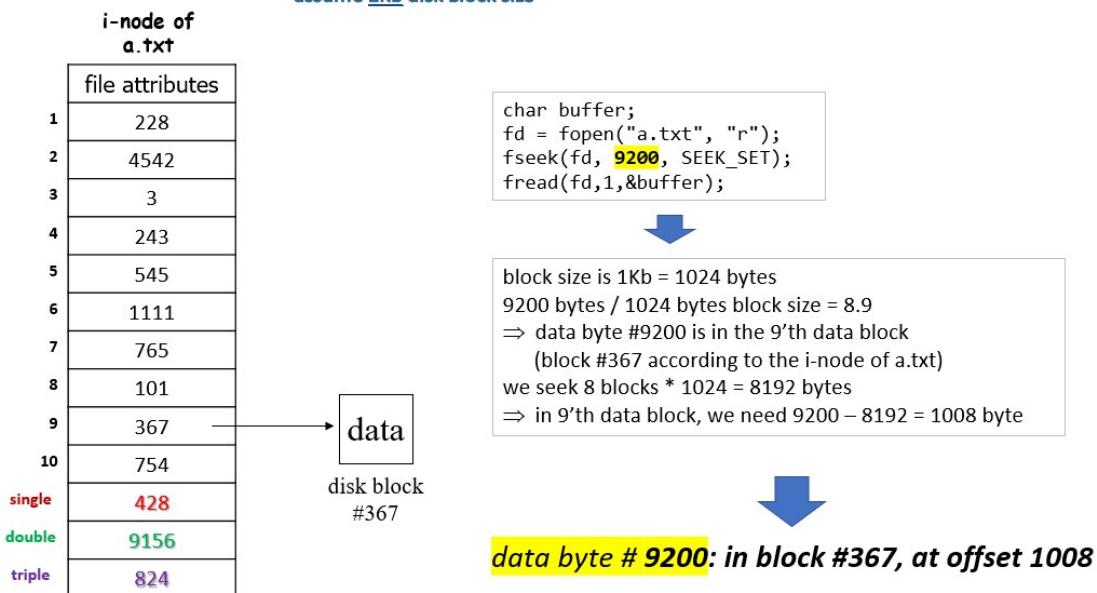
Unix: i-node structure

assume 1Kb disk block size



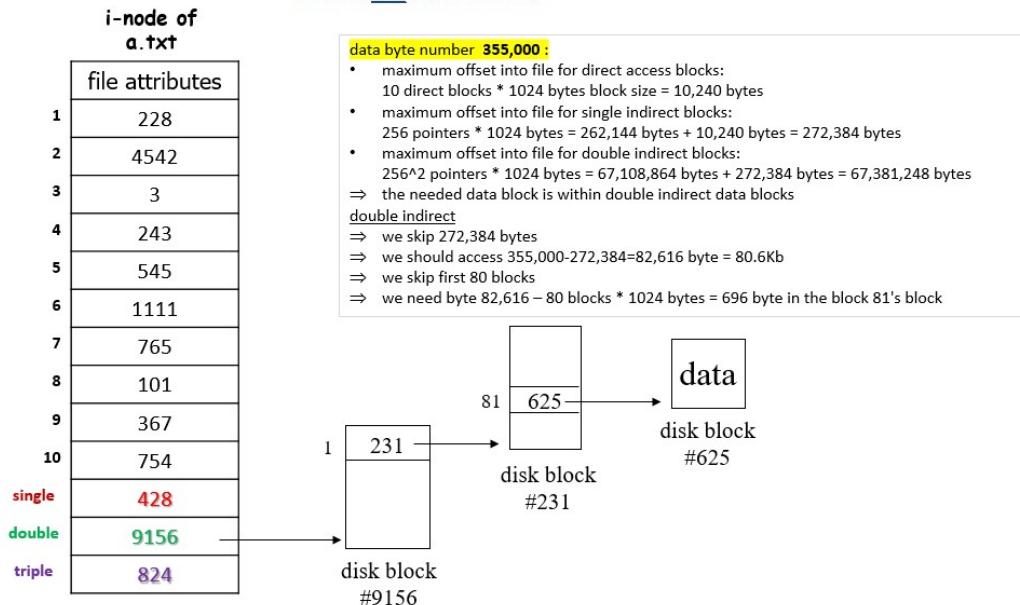
Unix i-nodes - counting bytes example

assume 1Kb disk block size



Unix i-nodes - counting bytes example

assume 1Kb disk block size



Unix FileSystem

מערכת הקבצים של יוניקס בנוייה בצורה שונה מ-FAT.

יצוג הקובץ

קבצים במערכת מיוצגים על ידי מבנה הקבצים inode. אנו נפשטו את מבנה מערכת קבצים זו על מנת להקל על לימודה.

ה-inode מחזיק בתיםعلילונים את תכונות הקובץ (לא כולל שם) ולאחר מכן מעיר של 10 כתובות ל-10 הבלוקים הראשונים של הקובץ בדיסק.

אם הקובץ תופס יותר מ-10 בלוקים, אז ישנו מצביע לטבלה אחרת עם 10 מצביעים נוספים. זה ה-single indirect. אם הדבר אינו מספיק, לאחר ה-single indirect ישנו מצביע לטבלה שמכילה triple indirect נוספת. אם גם זה לא מספיק, ישנו מצביע לטבלה המכילה double indirect נוספים ונקראת triple indirect.

נניח שאודל הבלוק בדיסק הוא 4Kb. כל רשומה node-i מכילה כתובות של עשרה הבלוקים הראשונים של הנתונים. כלומר, קובץ יכול לאחסן 10 בלוקים * 4Kb = 40Kb של נתונים. אנו מחלקים את הקובץ לבלוקים של נתונים, כל אחד בגודל 4Kb. ישנו $1024 / 4 = 256$ מצביעים ישירים לבלוקים של נתונים. הקובץ יכול לאחסן עד $1024 \text{ blocks} * 4\text{Kb} = 4\text{Mb}$ לכל בלוק = 4Mb של נתונים. ישנו $1024 / 4 = 256$ מצביעים כפולים לבלוקים של נתונים. הקובץ יכול לאחסן עד $1024 * 4\text{Mb} = 4\text{Gb}$ של נתונים.

במערכת הקבצים של יוניקס, כל קובץ משירט לבנייה נתונים בשם node-i. ה-node-i מכיל מידע על הקובץ, כמו תכונות וכתובות דיסק של בלוקים של הקובץ. באמצעות ה-node-i, אפשר למצוא את כל הבלוקים של הקובץ. היתרון הגדול של שיטה זו מעל קבצים מקוררים באמצעות טבלה בזיכרון הוא שה-node-i צריך להיות בזיכרון רק כאשר הקובץ המתאים פתוח. אם כל node-i תופס א בתים ומקסימום של K קבצים יכולים להיות פתוחים בו זמן נתון, היזכרון הכלול שהמערך המחזיק את ה-nodes-i עבר הקבצים הפתוחים תופס הוא רק א בתים. רק כמה מהן זו של זיכרון צריכה להיות שמורה מראש.

אחת הביעות עם nodes-i היא שאם לכל אחד יש מקום למספר קבוע של כתובות דיסק, מה קורה כאשר קובץ גדול מעבר למגבלת זו? אחת הפתרונות היא לשמר את הכתובת الأخيرة של הדיסק לא בлок נתונים, אלא כתובות של בлок המכיל עוד כתובות של בлокים של דיסק. גרסאות מתקדמות יותר יכולות להכיל שני או יותר בлокים של כתובות או בлокים שמצוירים לבlokים אחרים של כתובות. הרעיון מומש במערכת הקבצים של יוניקס וגם במערכת הקבצים NTFS של Windows.

נניח שאודל הבלוק בדיסק הוא Kb. בואו נחשב את הגודל המרבי של קובץ במערכת הקבצים של יוניקס, כאשר גודל הבלוק הוא Kb. יש לנו 10 בлокים ישירים * Kb = 10Kb = 1 Kb של בлок מצבייע ישיר / 4-byte לכל גודל מצבייע לבlok = 256 בлокים מצבייעים ישירים של Kb = 256Kb = 1 Kb נוספים של נתונים. יש לנו Kb של בлок מצבייע כפול / 4-byte לכל גודל מצבייע לבlok = מצבייעים ל-256 בлокים מצבייעים ישירים * Kb = 256 Kb של נתונים לכל אחד = 64Mb נוספים של נתונים. יש לנו Kb של בлок מצבייע משולש / 4-byte לכל גודל מצבייע לבlok = מצבייעים ל-256 בлокים מצבייעים כפולים * Kb = 64Mb של נתונים לכל אחד = 16Gb נוספים של נתונים.

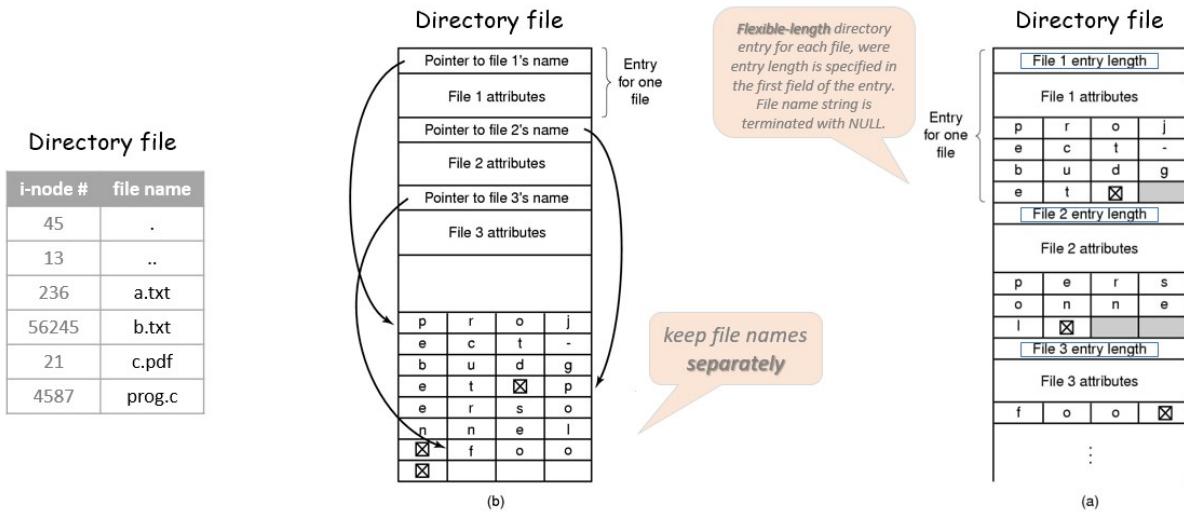
Implementing Directories

directory is a special-structured file

Directory file

i-node #	file name
45	.
13	..
236	a.txt
56245	b.txt
21	c.pdf
4587	prog.c

Supporting long file names



מימוש תיקיות

לפנינו שניثنן לקרוא קובץ, יש לפתחו אותו. כאשר קובץ נפתח, מערכת הפעלה משתמשת בשם הנתיב שסופק על ידי המשמש כדי לאתר את רשותת התיקייה על הדיסק. רשותת התיקייה מספקת את המידע הנדרש למציאת בלוקי הדיסק. בעוד שהדבר תלוי במערכת, באופן כללי מידע זה כולל את הכתובת של הקובץ כלו (עם הקצה רציפה), מספר הבלוק הראשון (שני התרחישים של רשימה מקושרת), או מספר ה-node-i. בכל המקרים, התפקיד העיקרי של מערכת התיקיות הוא למפות את שם הקובץ של ASCII למידע הנדרש לאתר את הנתונים.

ונואנו נספה הוא איפה יש לאחסן את התוכנות. כל מערכת קבצים מתחזקת מאפייני קובץ שונים, כמו בעל הקובץ וזמן היצירה של כל קובץ, והם חייבים להישמר במקום כלשהו. אפשרות ברורה אחת היא לאחסן אותם ישירות ברשותת התיקייה. כמה מערכות עושות לבדוק את זה. בעיצוב הפשטוט הזה, תיקייה מורכבת מרשימה של רשומות בגודל קבוע, אחת לכל קובץ, המכילה שם קובץ באורך קבוע, מבנה של מאפייני הקובץ, וכתוות אחת או יותר של דיסק (עד למקסימום מסוים) שמספרת איפה הבלוקים של הדיסק נמצאים.

עבור מערכות שימושות ב-nodes-i, אפשרות נוספת לאחסן התוכנות היא ב-nodes-i, ולא ברשותת התיקייה. במקרה זה, רשותת התיקייה יכולה להיות קצרה יותר: רק שם קובץ ומספר node-i. כפי שנראה מאוחר יותר, לשיטה זו יש כמה יתרונות על פני השימוש ברשותת התיקייה.

עד כה הנקנו באופן מושג של קבצים יש שמota קצרים באורך קבוע. ב-SOS-MS לקבצים יש שם בסיס של 8-17 תווים ורחבה אופציונלית של 1-3 תווים. ב-UNIX גרסה 7, שמota הקבצים היו 1-14 תווים, כולל כל ההרחבות. עם זאת, כמעט כל מערכות הפעלה המודרניות תומכות בשמות קבצים ארוכים יותר, באורך משתנה. איך אפשר למשוך את זה?

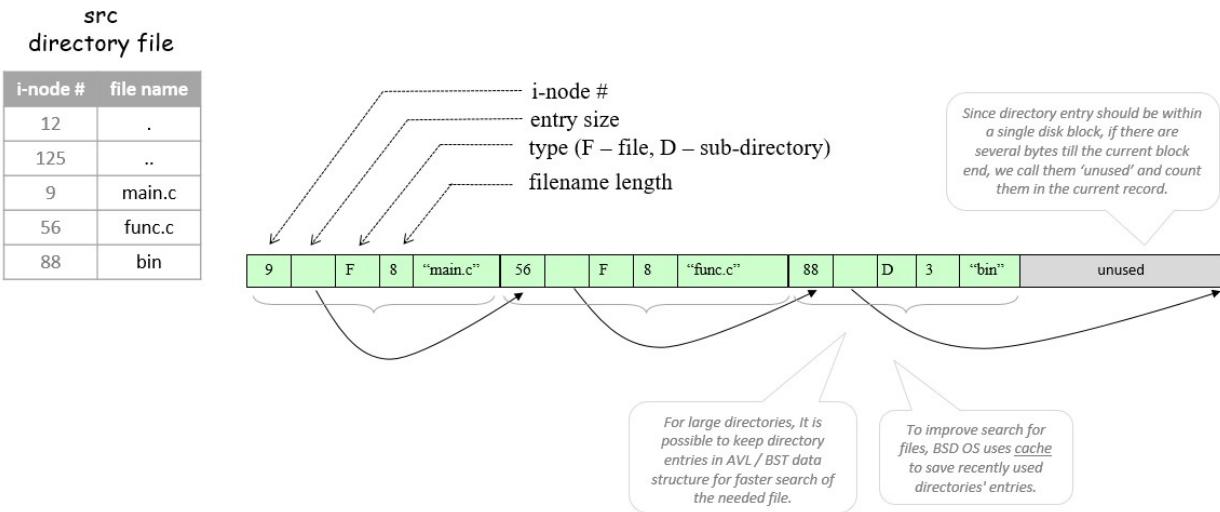
תמייה בשמות ארוכים:

גישה פשוטה ביותר היא להגדיר מגבלה על אורך שם הקובץ, בדרך כלל 255 תווים, ואז להקצות 255 תווים שמורים לכל שם קובץ. גישה זו היא פשוטה ואף אפשרית חיפוש בצורה קלה יותר, בפועל היא מבזבזת הרבה

מקום בתיקייה, מכיוון שמעט קבצים יש להם שמות כל כך ארוכים. מבנה שונה משמש ברוב המערכות. לא נשמר את שם הקובץ בצדם לתכונות האחרות של הקובץ. אנחנו בעצם נרצה את כל השמות למקום אחד ונשмар ב-i-node מצבייעים לשם. כל רשומה מתחילה עם מספר בתים שמציאן את אורך הרשומה. אחרי מגע שם הקובץ, שמתייעים ב-TNULL. התווים הנוגעים ברשומה משתמשים לשמירה על מספר ה-i-node- או כתובת הדיסק, תלוי במערכת.

BSD Unix Directory Entry Structure

Example: directory "src" with 3 entries - "main.c" , "func.c" , and subdirectory "bin"

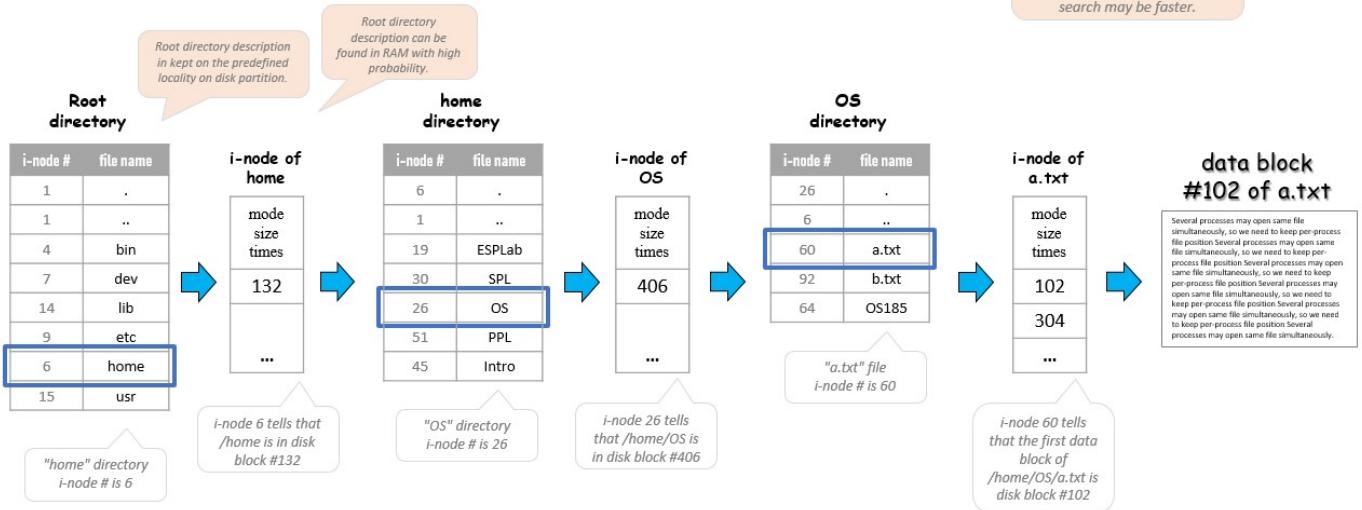


מבנה רשומת תיקייה של Unix BSD

דוגמה: תיקייה "src" עם 3 רשומות - "c", "main.c", "func.c", ותת-תיקייה "bin".
لتיקיות גדולות, אפשר לשמור רשומות תיקייה במבנה נתונים של AVL / BSTデータ structure ליחישוש מהיר של הקובץ הנדרש.
כדי לשפר את החישוש של קבצים, מערכת הפעלה BSD משתמשת במתמן לשמירה על רשומות תיקיות שנמצאות בשימוש לאחרונה.
אחרי רשומת התקייה צריכה להיות בתוך בלוק דיסק אחד, אם יש מספר בתים עד לסוף הבלוק הנוכחי, אנחנו קוראים להם לא בשימוש ומחשיבים אותם ברשומה הנוכחיית.

BSD Unix Directory search example

Example: the steps needed for looking up "/home/OS/a.txt"



דוגמה לחיפוש בתיקיה של BSD Unix

דוגמה: השלבים הנדרשים לחיפוש "/home/OS/a.txt"

אם אנחנו מקבלים נתיב ייחסי, לאחר שהתחילה יודע את node-i של התיקיה הנוכחיית שלו, החיפוש שלנו יוכל להיות מהיר יותר.

תיקיית השורש נשמרת במקום מוגדר מראש בחלוקת הדיסק ויכול להימצא ב-RAM בהסתברות גבוהה.

שאלה: בהינתן הנקודות אלו, כמה גישות לדיסק יש עבור פונקציה שפותרת את הקובץ וקוראת ממנו בית אחד?

תשובה:

גישה 1 - גישה ל-node-i של root

גישה 2 - ל-root עצמו.

אחר root היא תיקיה אז נוכל לאתר מתוכה את node-i של home.

גישה 3 - מציאת node-i של home. שם רשום איפה node-i של התיקיה נמצא.

גישה 4: הולכים ל-block data כדי להגיע לתיקיה של home.

גישה 5: ל-node table למציאת node-i כדי למצוא את הקובץ המציג את התיקיה OS.

גישה 6: גישה נוספת ל-node-i.

גישה 7: גישה ל-block הראשון של a.txt.

במקרה האروع ביותר הוא indirect triple indirect וצריך עוד 3 כדי להביא את הבלוק הראשי.

עוד גישה נוספת לבlok של data.

יש עוד גישות שאנו לא יודעים - אבל זה התשובה זה לא ידוע. הייתה לנו הנחה סטטיסטית שאנו יודעים איפה home. מי אמר שהוא תופס בלוק אחד? צריך בשבייל זה להניח שקובץ תיקיה תופס בלוק אחד.

אפשר לזרות לפי החישוב ישיר שלנו, אבל לא יודעים באיזה בלוק נמצא המידע. יודעים מה המסלול אבל לא יודעים מה כתוב על המסלול.

חידוד: רוט - לא בהכרח במיקום קבוע. גישה ל-node-i של root - קבוע

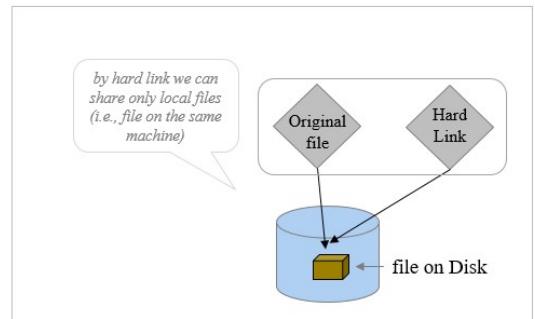
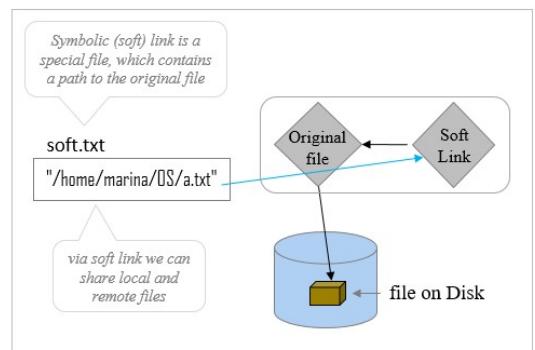
Shared Files - Links

```
marina@vm:~/OS$ ls -s a.txt soft.txt
marina@vm:~/OS$ ls a.txt hard.txt
marina@vm:~/OS$ ls -li
total 24
655599 -rw-rw-r-- 3 marina marina 3 May 9 20:20 a.txt
655599 -rw-rw-r-- 3 marina marina 3 May 9 20:20 hard.txt
655609 lrwxrwxrwx 1 marina marina 5 May 30 23:16 soft.txt -> a.txt
655425 drwxrwxr-x 5 marina marina 4096 Jan 15 15:32 SPL
655416 drwxrwxrwx 6 marina marina 4096 May 30 18:07 SPLab
```

i-node #
file type
(d = directory, l = soft link)

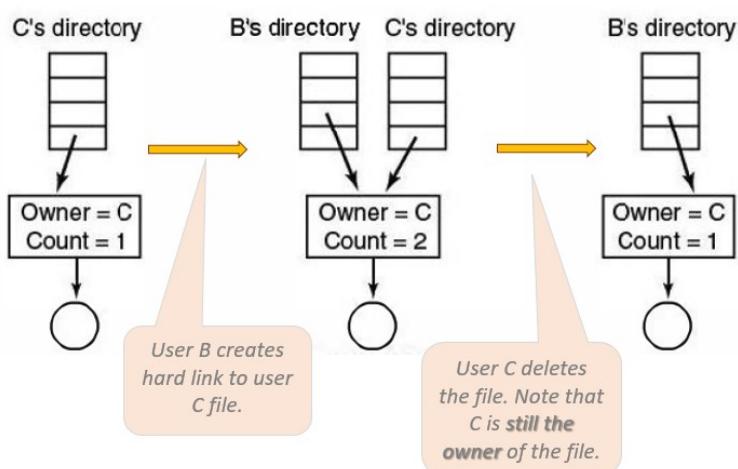
file name	i-node #
a.txt	655599
SPL	655425
SPLAB	655416
soft.txt	655609
hard.txt	655599

soft link has its own i-node #, since it is separate file
hard link has the same i-node # as the original file, since hard link is just an additional reference to the original file



Issues with linked files

- Files manage reference counting (of hard links) for correct deletion.
This may lead to problems.



Suppose user C has some limited quota on Disk, and C wants to delete some large file to free space on their quota. Since there is still a hard link to this file, even if C deletes this file, the file would not be deleted indeed...

So, deletion of your own files might depend on other users...

קבצים משותפים - קישורים ביונייקס

דיברנו בעבר כי ניתן ליצג מערכת קבצים כעוצם מכוון וחסר מעגלים - דבר המאפשר שיתוף קבצים. אך הדבר מביא אותנו מרכובות חדשות ואתגרים נוספים, אך הראשית היא מה לעשות כאשר רוצים להצמיד בЛОקדים חדשים של מידע לסוף הקובץ על ידי קובץ אחד, במצב אשר מסווה את השינויים אחרים.

לשם כך ישנו 2 דרכים להתמודדות עם הבעיה, או במילים אחרות: 2 דרכים שונות לשיתוף קבצים ביונייקס: קישור רך (Soft Link) ו קישור קשיח (Hard Link).

Soft Link aka Symbolic Link

ברגע שמשתמש רוצה לknob קשור לקובץ של משתמש אחר, מערכת הקבצים מיצרת קובץ חדש LINK מסוג SYMBOLIC בתיויקית

המשתמש, אשר מכיל רק את הנתיב לקובץ. כאשר משתמש מעוניין לקרוא את הקובץ, מערכת הפעלה מזזה כי מדובר בקובץ LINK, מאתרת את הקובץ המקורי ואוז קוראת מתוכו.

מכאן, הקישור מכיל מספרinode- ייחודי, לאחר שהוא מצביע לקובץ נפרד, ובכך מאפשר שיתוף של קבצים מרוחקים.

Hard Link:

בגישה זו, הLINK בפועל לא רשום ברשימת התקינה אלא משוייר לבנייה הקבצים המקוריים. כאן, רשימת התקינה עצמה מחזיקה מצביע לבנייה הנתונים. ביוניקס, הדבר בא לידי ביטוי בכך שרשות התקינה מתווספת רשומה חדשה ובה מספרinode של הקובץ המקורי.

בעיות אפשריות:

שתי האפשרויות מביאות איתהן חסכנות. באישה השנייה אם בעל הקובץ מוחק את הקובץ, אין ניתן לחפש ולמחוק את כל רשומות התקינה שמצויבות לקובץ? אפשר לפטור את זה על ידי מחיקת רשומות התקינה של הקובץ אך שמירה על inode-ו עצמו, אבל אז עולה בעיה אחרת.

נניח שיש שני אנשים שעובדים על פרויקט מול שרת: אחד יוצר קישור קשיח והשני יוצר את הקובץ המקורי. אם הפרויקט מסתיים והאדם שייצר את הקובץ המקורי מוחק את הקבצים שלו, בגלל קישור הקשיח, הקבצים בפועל לא ימחקו. זה יוצר מצב שבו הקובץ נמחק אך בפועל הוא לא מתעדכן בדיסק.

דוגמה נוספת: מערכת קבצים מנהלת ספירת הפניות (של קישוריםים קשיים) למחיקה נכון. זה עשוי להוביל לביעוות. נניח שימוש B יוצר קישור קשיח לקובץ של משתמש C. משתמש C מוחק את הקובץ. נשים לב ש-C עדין הוא בעל הקובץ. נניח שהשימוש C יש מכסה מגבלת על הדיסק, ו-C רוצה למחוק קובץ אדול כדי לשחרר מקום במכשירו. לאחר מכן יש קישור קשיח לקובץ זה, אפילו אם C מוחק את הקובץ, הקובץ לא ימחק באמת.

הקישור הסמלי נמנע מבעה זו, לאחריהם מספקים כתובות למשתמשים ורק לבאים האמיתיים של הקובץ יש קישור ישירinode-. כאשר הבעלים של הקובץ מוחק את הקובץ, הוא נמחק למעשה וכל ניסיון לגשת אליו דרך קישור סמלי נכשל. עם זאת קישור הסמלי מביא אליו דרישת גישה נוספת לדיסק על מנת לאתר את מקום הקובץ.

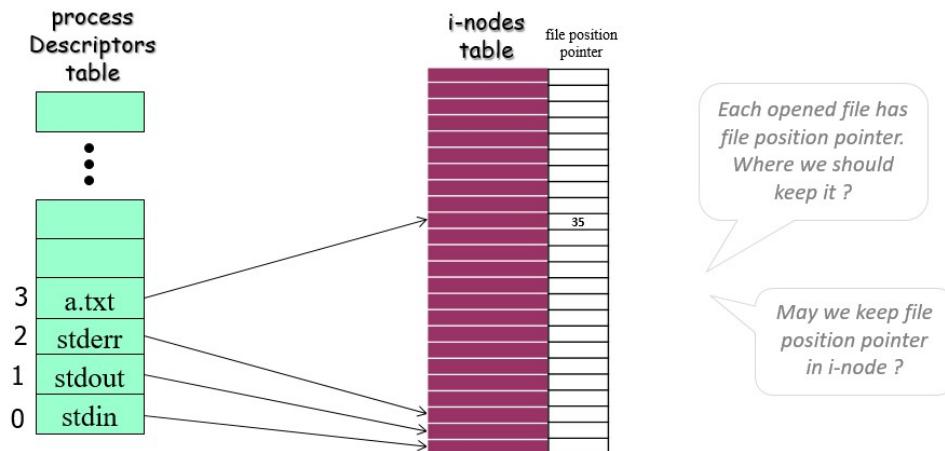
ניתן לראות זאת בדוגמה לעיל: בקובץ שמייצג התקינה, הקובץ בשם soft.txt יהיה בעל מספרinode- שונה מהמקורו, וההרשאות של הקובץ יתעדכנו בהתאם. בקישור קשיח, לא נוצר קובץ חדש, אלא מתווסף לקובץ שמייצג את התקינה אלמנט חדש שהinode- שלו זההinode- של הקובץ המקורי.

בעיה נוספת אשר הקישורים מ-2 הסוגים מביאים איתהן, הוא קיום של מספר נתיבים לאותו הקובץ. תוכניות שسورקות תיקיות ותאי תיקיות עלולים להתקלק בקבצים מקושרים מספר פעמים, דבר אשר עלול להוביל לשכפול קבצים כאשר מבצעים גיבוי או העתקת מידע.

הסוגיות הללו מחייבות סיבה אמיתית להשתמש בקישור קשיח ושימוש עדין בו.

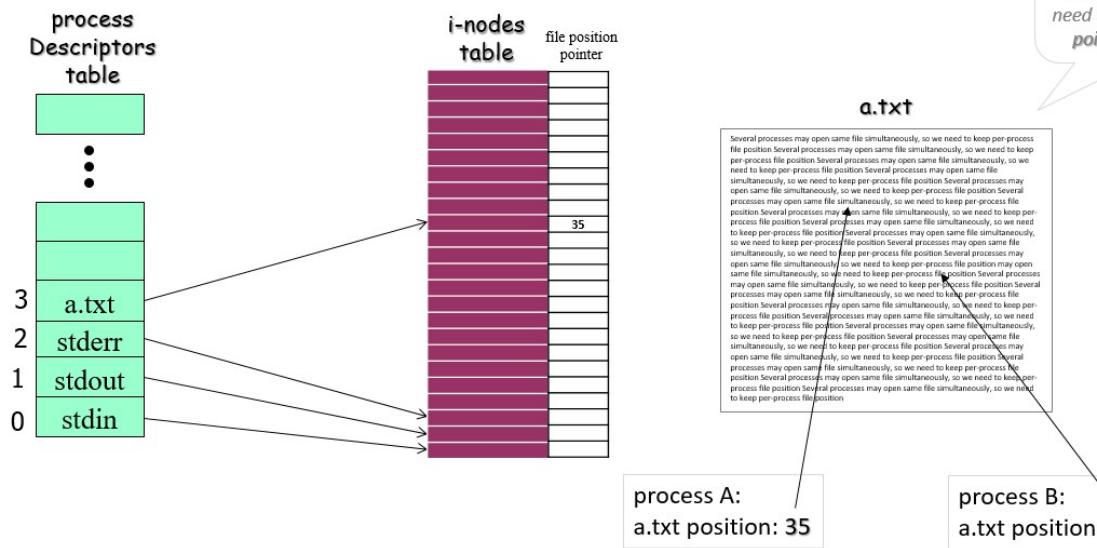
File descriptors table take 1

contains information about files opened by the process



File descriptors table take 1

contains information about files opened by the process



טבלת מזהה קבצים - ניסיון ראשון

ברגע שתהיליך פותח קובץ, מערכת הפעלה יוצרת עבורו מזהה קובץ (fd) וגם מצביע למקום בקובץ. עכשו נרצה לדון בשאלת איפא כדי למערכת הפעלה לשמור את מצביע המיקום בקובץ.

במעבדות למדנו שב-PCB יש טבלה שנקראת Descriptors table. כברירת מחדל, כאשר מערכת הפעלה מעלה תוכנית, היא פותחת את stdio. בדוגמה שלנו, התחילה גם פותח את הקובץ txt.a. ראיינו גם שת'דים ותהליכי בן-

טבלת מהי הקבצים מכילה מידע אודוט הקבצים שנפתחו על ידי התהילר. לכל קובץ שנפתח יש מצביע למיקום בקובץ, השאללה היא. איפה כדאי לנו לשמר את מצביע המיקום בקובץ?

הצעה ראשונה (שאיינה תקנית):

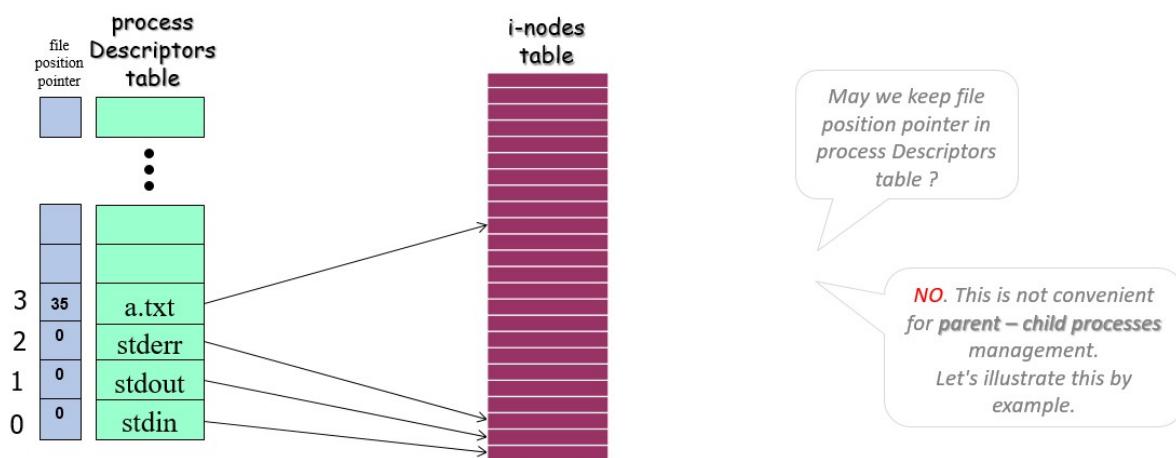
נשמר מבנה נתוני משותף שבו לכל קובץ יש מצביע למיקום בקובץ. הבועה הראשונה היא שאמם כמה תהליכיים ירצו לקרוא מקובץ, אנו רוצים שהם יוכלו לעשות זאת ממיקומים שונים. אך אז, מה נשים בתיעוד של מצביע המיקום בקובץ.

אי אפשר לשמור את מצביע המיקום בקובץ ב-epoch-i, מאחר שמספר תהליכיים יכולים לפתח את אותו הקובץ בו זמן-nit, ולכן אנחנו צריכים לשמור את מצביע המיקום בקובץ לכל תהליך.

לדוגמא, תהליך A יכול להיות במיקום 35 בקובץ txt.a, בעוד שתהליך B יכול להיות במיקום 64 באותו הקובץ.

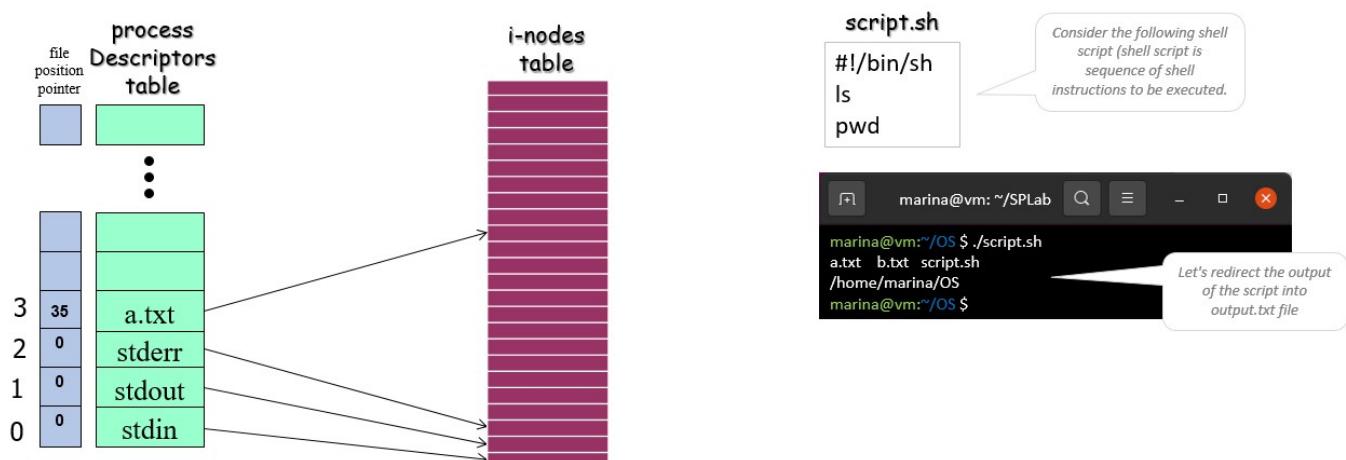
File descriptors table take 2

contains information about files opened by the process



File descriptors table take 2

contains information about files opened by the process



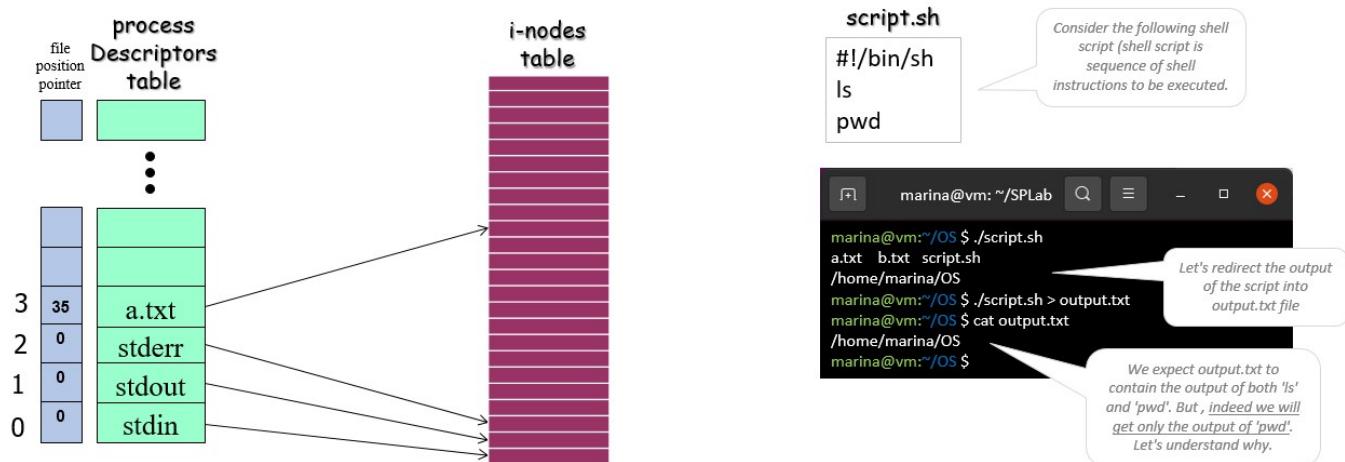
טבלת מזהה קבצים - ניסיון שני

האם ניתן לשמור את מצביע המיקום בקובץ בטבלה descriptors של התהילר? התשובה היא לא. זה לא נכון לניהול תהליכי של הורה ליד.

בואו נדגים את זה בדוגמה. נניח שיש לנו את הסקRYPT הבא שמריץ רצף של הוראות shell. אנחנו מנהלים את הסקRYPT ומנתבים את הפלט שלו לקובץ output.txt. אנחנו מצפים שהקובץ output.txt יוכל את הפלט של שני הפקודות 'ls' ו'pwd'. אך בפועל, קיבל רק את הפלט של 'pwd'.

File descriptors table take 2

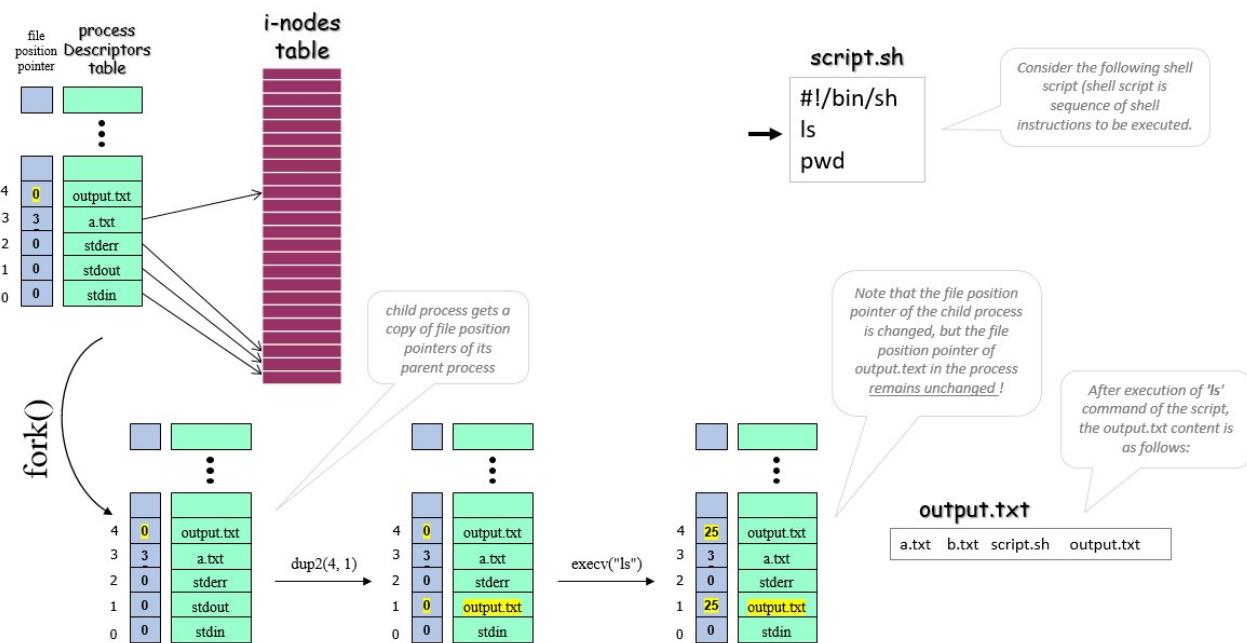
contains information about files opened by the process



הערה: התמונה של הטרמינל עבר עריכה ידנית כדי להתאים למודל המתואר (ברור שבטרמינל בלינוקס זה יעבוד בבדיקה כמו שאנו מצפים שהוא יעבוד)

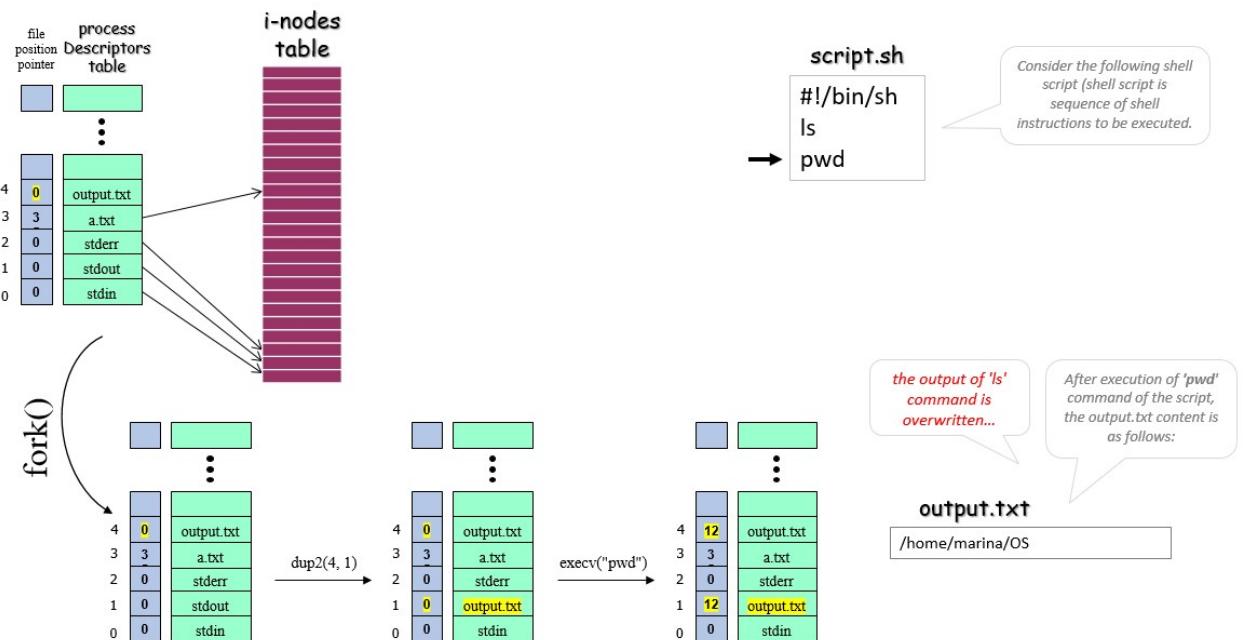
File descriptors table take 2

contains information about files opened by the process



File descriptors table take 2

contains information about files opened by the process



באו נזכיר למה. לאחר שה-shell מבצע `fork`, תהליך הילד מקבל עותק של מצביע המיקום של התהיליך ההורה. שימו לב שמדובר במיקום של תהליך הילד משתנה, אך מצביע המיקום של הקובץ `output.txt` בטהילך ההורה לא משתנה!

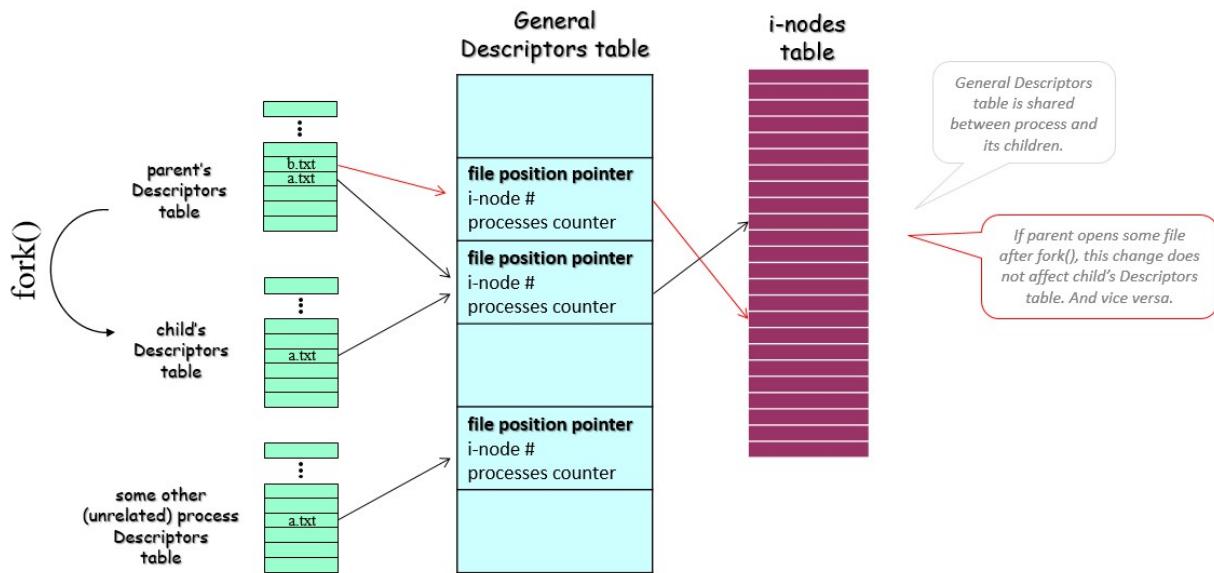
לאחר ביצוע הפקודה `'ls'` של הסקריפט, התוכן של `output.txt` הוא כדלקמן: הפלט של הפקודה `'ls'` נמחק ולאחר מכן ביצוע הפקודה `'pwd'` של הסקריפט, התוכן של `output.txt` הוא כמוופיע לעיל

זה קורה מכיוון שרגע לפני shell מבצע `exec`, הוא צריך להחליף את `stdout` בקובץ הרצוי. הוא משכפל את ה-`fd` ובמקום `stdout` יהיה הקובץ הרצוי. כאשר הפקודה `'ls'` מתחבצת, היא ממשיכת קרגיל ומדפיסה לקובץ כמוilo זה היה ה-`stdout`. ה-`position` מתעדכן במהלך הכתיבה לקובץ על ידי `'ls'`. כתוצאה לכך, כאשר ה-`shell` מרים את

'pwd', הוא כותב לאוטו מקום שאליו הצביע ה-'\$', ולא למקום שאליו הצביע ה-'\$'ו, ולכן הוא דורס את התוכן!

File descriptors table take 3

contains information about files opened by the process



טבלת מזהה קבצים - ניסיון שלישי (עובד)

בניסיון שלישי זה, אנו מציעים לשמר טבלת descriptor כללית המשותפת לתהליכי ההורה והילד. בעת, כאשר תהיליך הילד מקדם את מצביע המיקום בקובץ, הוא גם מקדם את מצביע המיקום בקובץ של תhilיך ההורה. זו הtegaות תקינה שאנו חנכו מזמן לה.

אם אנחנו רוצחים להפסיק את ההתקדמות המשותפת הזו, אנחנו יכולים לאפשר לתhilיך ההורה לסגור את הקובץ, לזכור את מיקום `h-position` `file`, ולאחר ה-`fork`, לסגור את הקובץ, לפתח אותו מחדש, ולהזיז את מצביע המיקום למיקום ששמרנו. בדרך זו, הקובץ לא מופיע בטבלה המשותפת. זה אפשרי רק אצל ההורה, מכיוון שרק ההורה יודע שיש לו תהיליך בן (התhilיך הבן לא יודע שיש לו הורה, אלא אם כן משתמשים בקריאה מערכת).

אם ההורה פותח קובץ מסוים לאחר ה-`fork()`, שינוי זה לא משפיע על טבלת descriptors של התhilיך הבן, ולהפך.

Block size

• Large blocks

- high internal fragmentation
- faster sequential access
 - less blocks to read/write – less seeks
 - we would use all the data in the requested block
- slower random access
 - larger transfer time, larger memory buffers
 - we would not use all the data in the requested block

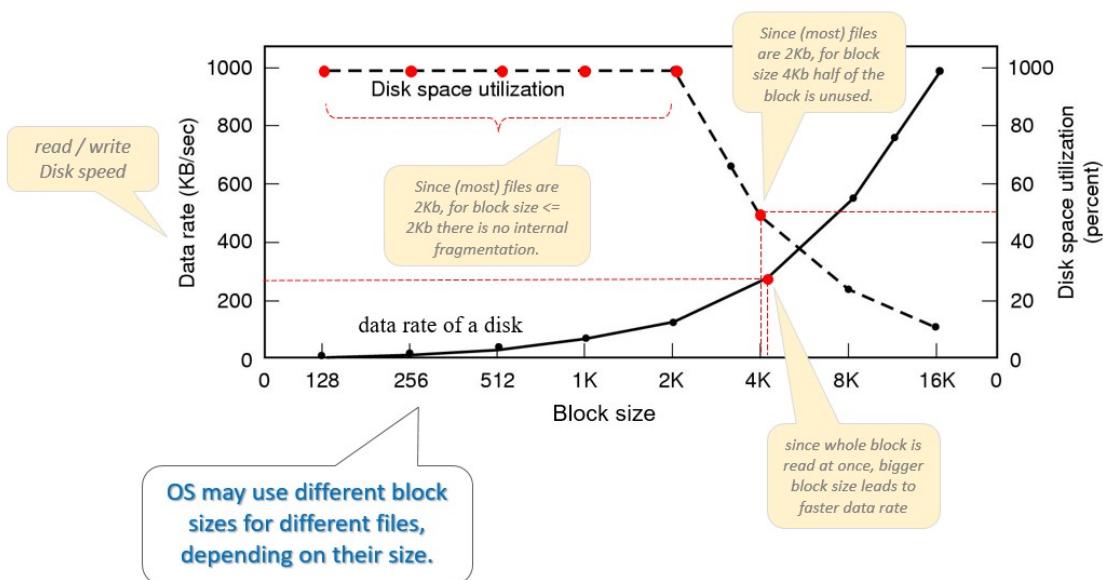
Last allocated block might be not fully used, thus causing internal fragmentation.

• Small blocks

- slower sequential access
 - more seeks
- faster random access
 - less data to bring from disk block

Block size considerations

assumption: most files are 2KB



ניהול ואופטימיזציה של מערכת קבצים

גודל הבלוק

כאשר אנו מדברים על גודל הבלוק במערכת קבצים, יש לנו שני אפשרויות כלליות: בלוקים גדולים ובלוקים קטנים. כל אחת מהן מזינה יתרונות וחסרונות מסוימת.

בלוקים גדולים מאפשרים גישה רציפה מהירה יותר, מכיוון שיש פחות בלוקים לקרוא או לכתוב, וכן פחות צורך בחיפושים. עם זאת, הם עלולים להוביל לפיצול פנימי גבוה יותר, כאשר לא נמצא שימוש לכל הנתונים בבלוק המבוקש. בנוסף, הם עשויים להאט את הגישה האקראית, מכיוון שזמן ההעברה הוא גדול יותר ודרושים מאגרי זיכרון גדולים יותר.

בלוקים קטנים, מצד שני, מאפשרים גישה אקראית מהירה יותר, כאשר יש פחות נתונים להביא מהבלוק של הדיסק. עם זאת, הם עשויים להאט את הגישה הרציפה, מכיוון שיש צורך בחיפושים רבים יותר.

שיקולים בנוגע לגודל הבלוק

הנחה נפוצה היא שרוב הקבצים הם בגודל של 2KB. לכן, עבור גודל בלוק של 2KB או פחות, אין פיצול פנימי. לעומת גודל בלוק של 4KB, חצי הבלוק לא מנוצל. מכיוון שכל הבלוק נקרא בביטחון אחד, גודל בלוק גדול יותר מוביל לשיעור נתונים מהיר יותר.

במחקר שנערך על התפלגות גודל הקובץ, נמצא שעבור בלוק בגודל 4KB, אחוז משמעוני של הקבצים יכולם להתאים לבלוק יחיד, ורוב הבלוקים בדיסק מנוצלים על ידי קבצים גדולים. זה מצביע על כך שגודל בלוק של 4KB הוא בחירה טובה למערכת קבצים ממוצעת.

Average time to access disk block

Disk access time parameters:

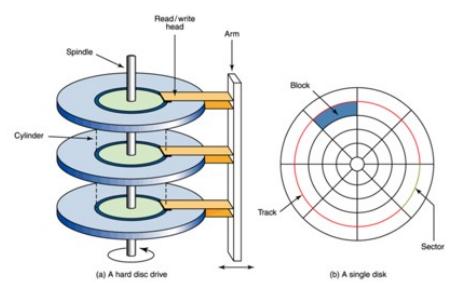
- seek time –time for head to get above a cylinder
- rotation time – time for disk to complete full rotation

Example:

- block size : 4 Kb
- average track size : 32 Kb
- average full track rotation time 8.33 ms
- average seek time 10 ms

tracks have different perimeters (sizes), so we suppose average track perimeter

in average, we move the disk arm $\frac{1}{2}$ way, i.e., $\frac{1}{2}$ of the disk radius



$$\text{Average time to access block: } 10 + \frac{8.33}{2} + \left(\frac{4 \text{ Kb}}{32 \text{ Kb}}\right) \times 8.33 \text{ ms}$$

average seek time – time to access a track that keeps the block we need

average time to get to the need block on a track (half rotation)

transfer time (time of reading block, relative part of needed rotation time)

זמן גישה ממוצע לבlok בדיסק

כאשר אנו מדברים על זמן גישה לבlok בדיסק, ישנו מספר פרמטרים שאנו צריכים לזכור בחשבון:

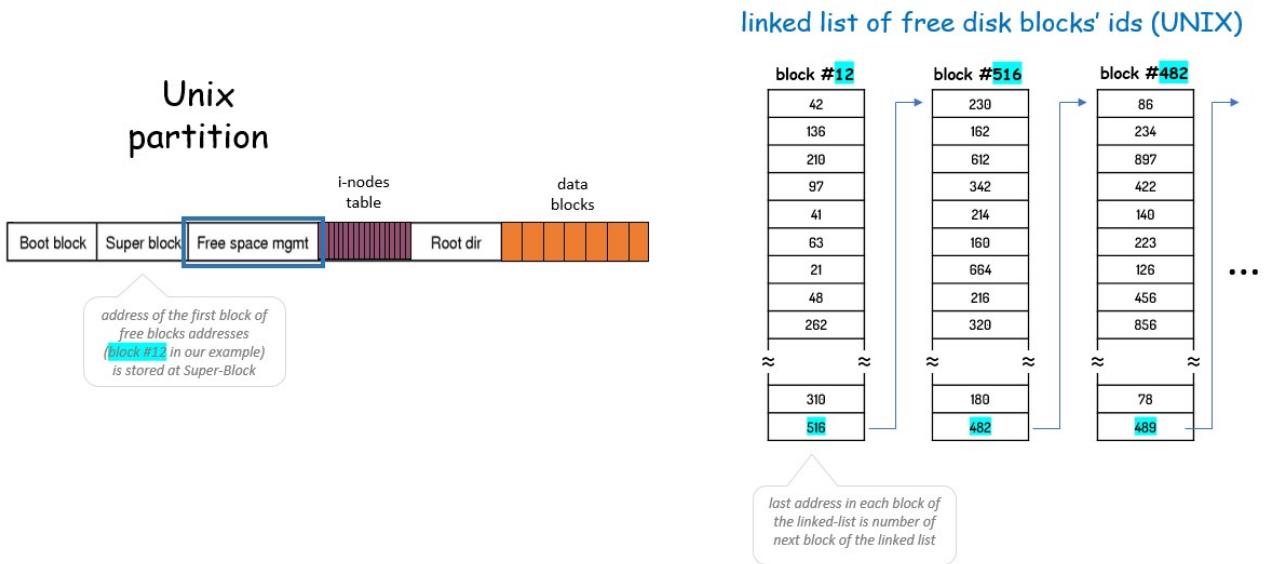
- זמן החיפוש (Seek Time) - הזמן שלוקח לראש הקריאה/כתיבה להגיע למעלג הנכון (Cylinder).
- זמן הסיבוב (Rotation Time) - הזמן שלוקח לדיסק להשלים סיבוב מלא.

נניח לדוגמה שגודל הבלוק הוא 4KB, גודל המסילה הממוצע הוא 32KB, הזמן הסיבובי המלא הממוצע הוא 8.33 מילישניות, וזמן החיפוש הממוצע הוא 10 מילישניות. במקרה זה, הזמן הממוצע לגישה לבlok הוא $10 + \frac{8.33}{2} + \left(\frac{4 \text{ KB}}{32 \text{ KB}}\right) \times 8.33 = 10 + 4.16 + 1 = 15.16$ מילישניות.

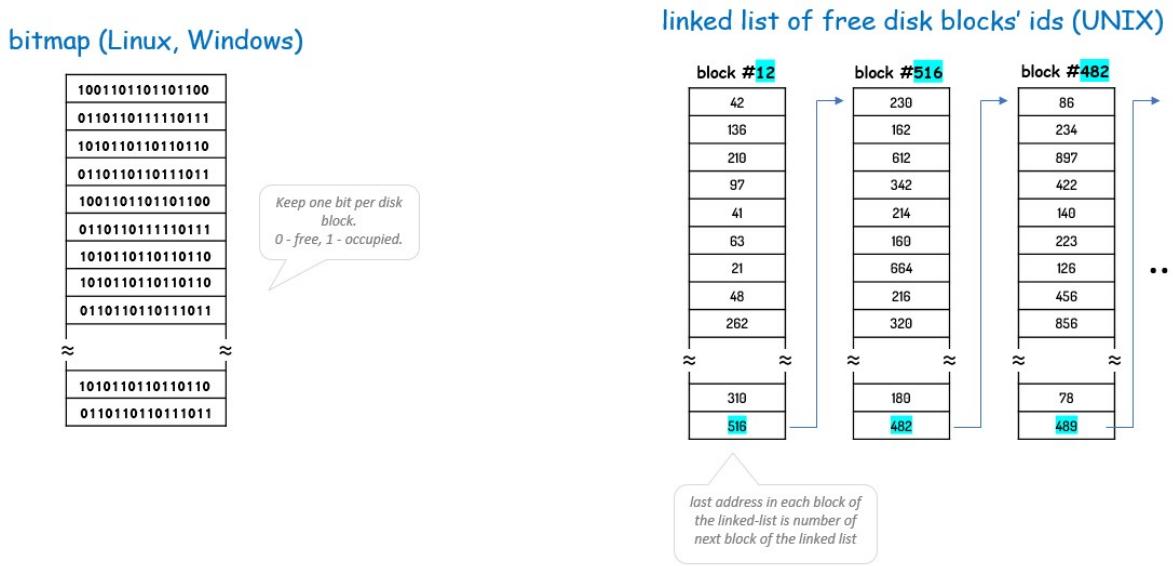
אנו מניחים שיש למסילות קוטרים שונים (גדלים), ולכן אנו מניחים את הקוטר הממוצע של המסילה. בממוצע, אנו מזינים את זרוע הדיסק חצי הדרך, ככלומר, חצי מהרדיסוס של הדיסק. הזמן החיפוש הממוצע הוא הזמן לגישה למסילה שמכילה את הבלוק שאנחנו צריכים. הזמן הממוצע להגעה לבlok החדש במסילה הוא זמן סיבוב חצי. הזמן ההעברה הוא הזמן של קריאת הבלוק, חלק יחסית של זמן הסיבוב החדש.

יש לשים לב כי החישוב היחסי של המסילה כאן הוא רק למטרות הדוגמה. במקרה הקצה שאנו מתעלמים ממנו, המסילה כל כך קטנה שהיא מכסה בלוק אחד. בפועל, שיעור הנתונים עולה באופן כמעט ליניארי לאורך הבלוק עד שזמן המעבר הופך להיות משמעותי.

Keeping track of free blocks



Keeping track of free blocks



ניהול בלוקים שאינם בשימוש

ישנן שתיים גישות עיקריות לנושא זה: `dcaps` ורשימות מקשורות. יוניקס השתמשה בגישת הרשימות המקשורות בעוד שוינדוס ולינוקס משתמשות בגישת ה-`bitmap`.

בגישת הרשימות המקשורות יש שימוש ברשימה מקושרת של בלוקים, כאשר כל חוליה מכילה כמה שיותר מזהה בלוקים פנויים שניתן להחזיק בחוליה. מזהה הבלוק האחרון ברשימה הוא מעשה הפניה לחוליה הבאה. לדוגמה,

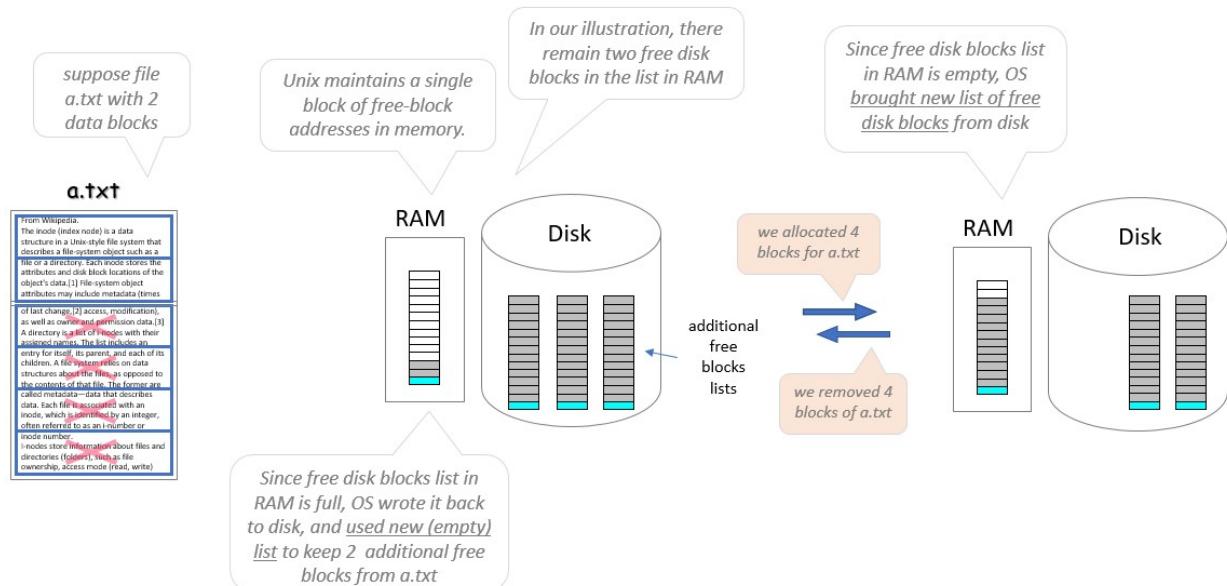
עם בלוקים בגודל 1KB ומספרם בבלוקים בגודל 32 ביט, כל חוליה (בלוק) יכול להחזיק לכל היותר 255 מזהה בלוקים פנויים. לאחסן הכתובות של כל הבלוקים הפנויים בדיסק בגודל 1TB (כמיליארד בלוקים) ביחס של 255 כתובות לבלוק, יש צורך ב-4 מיליון בלוקים. מכאן ניתן לראות כי השיטה דורשת כמה שטוחה יותר של שטח אחסון, אך היא עילה לניהול בלוקים חופשיים כאשר הם מגיעים לאחר מספר בלוקים צמודים.

בגישה השנייה יש שימוש ב-**bitmap** כאשר כל בית מיצג בלוק בדיסק: בלוקים פנויים מיוצגים על ידי 1-ים במאפה ובבלוקים מוקצים עם 0-ים. עבור דיסק בגודל 1TB, השיטה דורשת מיליארד ביטים, שהם 130 אלף בלוקים בגודל 1KB כל אחד. היא דורשת פחות מקום בדיסק ביחס לאיישת הרשימת המוקושרת מאחר שהיא משתמשת רק בבית 1 לכל בלוק. עם זאת, שיטה זו פחות יעילה כאשר הדיסק כמעט מלא, כאשר סכמת הרשימה המוקושרת תדרוש פחות בלוקים מה-.bitmap.

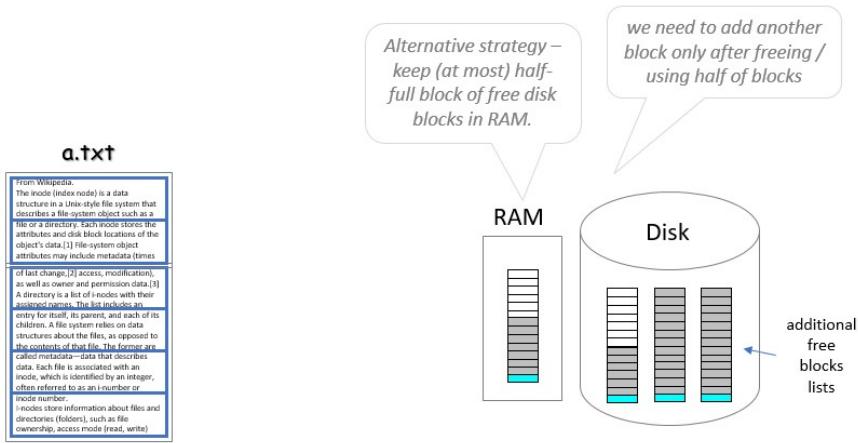
ניתן למשג את גישה זו בעזרת בלוק אחד בזיכרון - ברגע שמערכת ההפעלה תצטרכ לאייש לביטים אחרים, היא פשוט תביא אותם מהדיסק.

בסוף דבר, בחירת השיטה המתאימה לניהול בלוקים פנויים תלויות בנתונים מהעולם האמיתי שאין זמינים עד שהמערכת נמצאת בשימוש - דבר המקשה לבחירת השיטה המתאימה.

Preventing free-block thrashing (Unix)



Preventing free-block thrashing (Unix)



Thrashing:

במימוש גישת הרשימות הקשורות, יש צורך רק בבלוק אחד של מצביעים בזיכרון. כאשר קבצים נוצרים או נמחקים, המרכיבים תנהל את הבלוקים של הרשימה בזיכרון וכתובו אותם בדיסק במידת הצורך. עם זאת כאשר בלוק המצביעים כמעט ריק, שימוש תדיר בקבצים זמינים עלול להגדיל את מספר פעולות ה-IO בדיסק.

בוניקס, כאשר ישנים בלוקים שלמים שמכילים את מזהה הבלוקים הפנויים. כאשר מחקנו קובץ שתפס 2 בלוקים, נרצה להחזיר אותו לרשימה. אם נעשה ייוו של מחיקה ויצירה, נctrar לgeschת כמה פעומים לדיסק. כדי למונע מצב זהה, נחייב רשימה של חצי מהבלוקים הפנויים. אם אם הקובץ יעשה ייוו, יש מספיק בלוקים לחתת ומספיק בלוקים לרשום.

נפרט:

יש לנו בראם בלוק אחד - כי אנחנו עובדים עם קבצים ואנו רוצים לחתת בלוק פנוי ולא לrhoץ כל פעם בדיסק כדי למצוא צזה. יש לנו בלוק מס' 18 שהוא פנוי, ויש בלוק 19 שהוא פנוי ואם נרצה לדעת איפה הבלוק הבא שפנוי אפשר להוציא מהרשימה, אם הבלוק הבא

יש לנו קובץ txt שאגדל ב-4 בלוקים, נתנו לו בלוק מס' 19, בלוק מס' 18 והרשימה התרוקנה. אז הלאנו לדיסק כדי להביא את החוליה הבאה כדי לדעת מה כתוב שם ולהוציא עוד 2 בלוקים. כתע אחרי שמחקנו את 2 הבלוקים, נרצה להחזיר לרשימה. בלוק 20 ו-21 פנויים, אבל יש עוד 2 פנויים. לכן ניצור חוליה חדשה ונכתב בה בחזרה שיש בה 2 בלוקים פנויים.

אם נעשה ייוו של מחיקה ויצירה, נctrar לgeschת כמה פעומים לדיסק. רצינו שזה לא יקרה על זה לשמורנו את בראם, ובכל במקרה זה קורה. זה ה-free block thrashing. בשביל זה נשתמש בדרך כלל לפיו לא לשמור רשימה של בלוקים פנויים, נחייב רשימה של חצי מהבלוקים הפנויים. אם אם הקובץ יעשה ייוו, אז יש מספיק בלוקים לחתת ומספיק בלוקים לרשום.

כמו חלוקה של טושים לסטודנטים, אם רק נחלק - סבבה. אם אחד אחד מבקש טוש והשני רוצה להחזיר 2. חצי תיק מלא - זה טוב. החלטה אנושית. אם ממש יהי רעים איתנו, אפשר לבקש חצי מהגודל של הרשימה. יש לנו 1024 מצביעים, אם תחילך רוצה חצי הוא יצטרך לקחת 512. אם הוא ירצה לשחרר, הוא הוא יוכל לשחרר 512 פעם ואז כל 512 בלוקים נ Kapoor לדיסק ולא כל 4 פעמים

File Systems : outline

- ❑ Concepts
- ❑ File system implementation
 - Disk space management
 - Reliability
 - Performance issues
- ❑ NTFS
- ❑ NFS

File System Reliability

- **Disk damage can be a disaster**
 - loss of permanent data
 - difficult to know **what** is lost
- Disks have Bad Blocks that need to be maintained
 - Hardware Solution: sector containing bad blocks list, read by controller and invisible to operating system; some manufacturers even supply *spare sectors*, to replace bad sectors discovered during use
 - Software Solution: OS keeps a list of bad blocks, and prevents their use
- For file systems that use a *FAT* – a special symbol for signaling a bad block in the FAT

We also may keep several copies of mostly important data on Disk. For instance, two copies of **FAT** are automatically kept.

אמינות

אמצעי האחסון הם חומרה, ולכן עשויים להיות בהם פגמים ושחיקה עם הזמן (במיוחד עם SSD). מכאן, מערכת הפעלה תרצה לשמור גיבויים של קבצים חשובים לה.

אז איך אנחנו יודעים אילו בלוקים פגומים? ישנן שתי גישות עיקריות לנושא זה:

א. פתרון חומרה: הדיסק מנהל סקטור מיוחד ששמור את הרשימה של הבלוקים הפוגמים. הסקטור נקרא על ידי הבקר ואין נראה למערכת הפעלה. ישנו יצרנים שספקים אפילו סקטוריים נוספים, שימושיים להחליף בלוקים פוגמים שנמצאים במהלך השימוש.

ב. פתרון תוכנה: מערכת הפעלה שומרת רשימה של בלוקים פוגמים, ומונעת את השימוש. במערכות קבצים שימושות ב-FAT, יש סמל מיוחד לסימון בלוק פגום ב-FAT.

בנוסף, ישנה אפשרות לשמור מספר עותקים של נתונים חשובים במיוחד על הדיסק. לדוגמה, שני עותקים של FAT משמרים אוטומטית.

File system consistency - blocks

- count number of block references in free lists and in files (list of blocks in all i-nodes)

Block number	Blocks in use	Free blocks
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15	1 1 0 1 0 1 1 1 1 1 0 0 1 1 1 0 0	0 0 1 0 1 0 0 0 0 1 1 0 0 0 1 1



עקביות

ניהול מערכת קבצים עקבית ואמינה הוא דבר קריטי ממספר היבטים. מערכת קבצים אמינה מבטיחה את שלמות המידע המאוחסן בדיסק, אחרת הדבר עלול לגרום לפגיעה במידע בדיסק ואף באובדן. כמו כן הדבר מבטיח כי המשתמשים יכולים לשמור על הזמינות והאמינות של הקבצים שלהם.

מערכת קבצים לא עקבית יכולה בסכנתה את היציבות והאמינות של כל המחשב. אם מערכת קבצים נשארת במצב לא עקבי כתוצאה מכינוי מחשב לא תקין או קriseה של המחשב אז הדבר עלול לגרום לשגיאות במהלך תהליך עליית מערכת הפעלה.

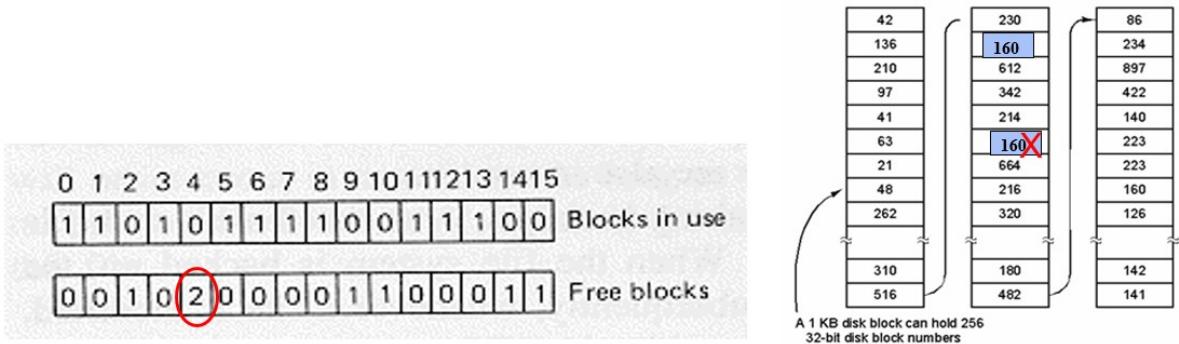
מערכת קבצים לא עקבית עלולה להכיל בלוקים שחסרים או בלוקים כפולים - דבר המוביל לשימוש לא יעיל באחסון. בלוקים חסרים גורמים לבזבוז שטח הדיסק, דבר אשר מוריד את הנפח הכללי הזמין לאחסון קבצים. בלוקים כפולים עלולים לגרום לבלבול ועלולים לגרום להשחתת מידע.

כדי להתמודד עם סוגיה זו, רוב המערכות מגיעות עם כלים כמו fsck ביוניקס או fs בוינדוס. הכלים הללו בוחנים את אמינות מערכת הקבצים ורקים לעתים קרובות במהלך תהליך ה-boot - במיוחד אחרי קriseה של המערכת.

ניתן לבצע זאת בעזרת מעבר על הבלוקים הפנויים ב-bitmap ובבלוקים בשימוש (בקב' שעוברים על כל ה-inodes ואוספים את כל הבלוקים שיש בהם שימוש) ואם רשום שהוא לא פנוי וגם מופיע בבלוקים בשימוש אז הרישום תקין.

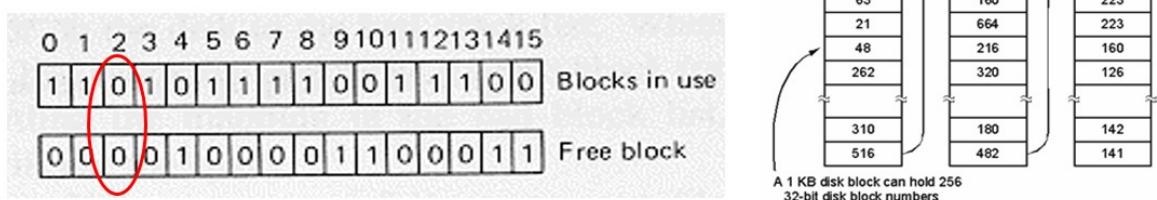
File system consistency - blocks

- count number of block references in free lists and in files
 - more than once in free list - delete all references but one



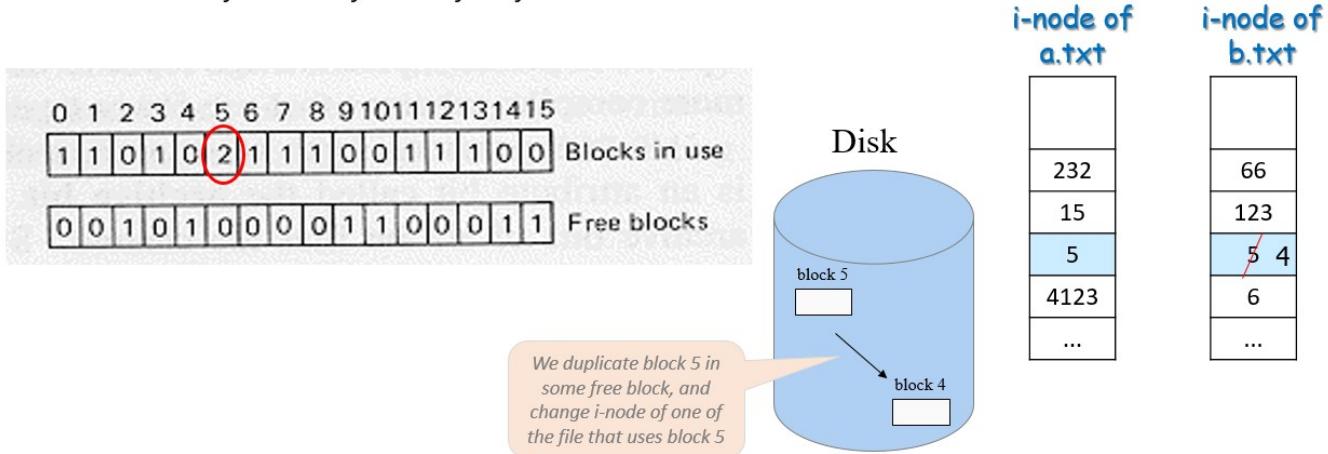
File system consistency - blocks

- count number of block references in free lists and in files
 - more than once in free list - delete all references but one
 - if both counts 0 - "*missing block*", add to free list



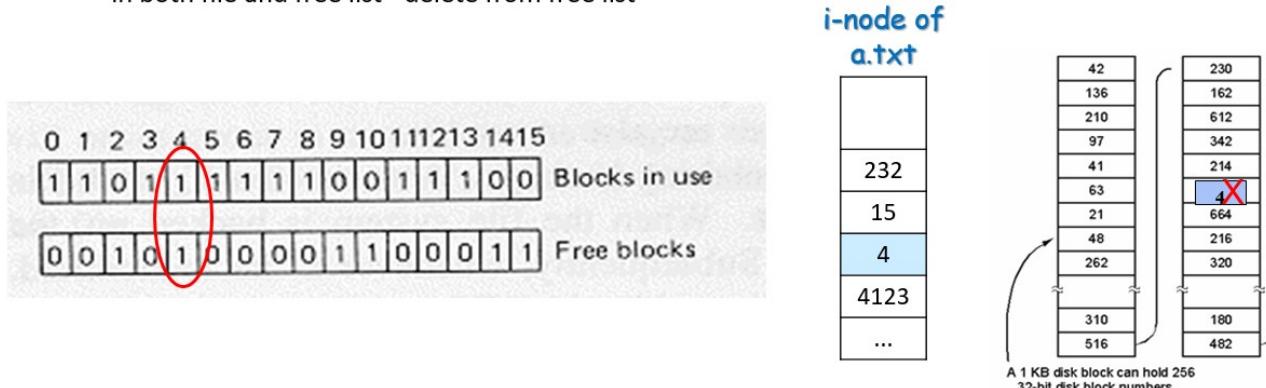
File system consistency - blocks

- count number of block references in free lists and in files
 - more than once in free list - delete all references but one
 - if both counts 0 - “**missing block**” – add to free list
 - more than once in files – **TROUBLE** – *duplicate block content to some other free block’ for one of the files a.txt or b.txt*



File system consistency - blocks

- count number of block references in free lists and in files
 - more than once in free list - delete all references but one
 - if both counts 0 - “**missing block**” – add to free list
 - more than once in files – **TROUBLE** – *duplicate block content to some other free block’ for one of the files a.txt or b.txt*
 - in both file and free list - delete from free list



בדיקה עקביות בложים:

בבדיקה זו נוצרות 2 טבלאות, כאשר כל אחת מהן מחזיקה קאונטר לכל בלוק ומאותחלות לאפס. הטענה הראשונה עוקבת אחר מספר הפעמים שבЛОק משמש לקובץ והטענה השנייה עוקבת אחר מספר הפעמים שבLOCK מופיע ברשימת הבלוקים הפנויים או ב-bitmap של הבלוקים הפנויים.

כל הבדיקה יעבור על כל ה-inodes בהתקן עצמו, ייצור רשימה של בלוקים אשר בשימוש בכל קובץ ויאדייל את ערכיו הקאונטרים בטבלה הראשונה בהתאם. כמו כן הכלי יבחן את רשימת הבלוקים הפנויים ויאדייל את ערכיו הקאונטר בטבלה השנייה בהתאם.

אם מערכת הקבצים עקבית, אז לכל בлок יהיה 1 רק בטבלה הראשונה (הблוק בשימוש) או בטבלה השנייה (הבלוק פנוי) אך לא בשנייהם.

דוגמה שלנו: מקרה אחד של פאגם יכול להיות שיפוי מספר 2 בראשית הבלוקים הפנויים (במילים אחרות, לא באמת bit אלא קאונטרים). במקרה זה, זה אומר שכאשר נרצה להשתמש בבלוק זה אנו עלולים להזכיר את הבלוק ל-2 קבצים שונים.

במקרה של בדיקת עקביות של המערכת, ברגע שמערכת הפעלה תראה את זה, היא תמחוק את המופיע הזה.

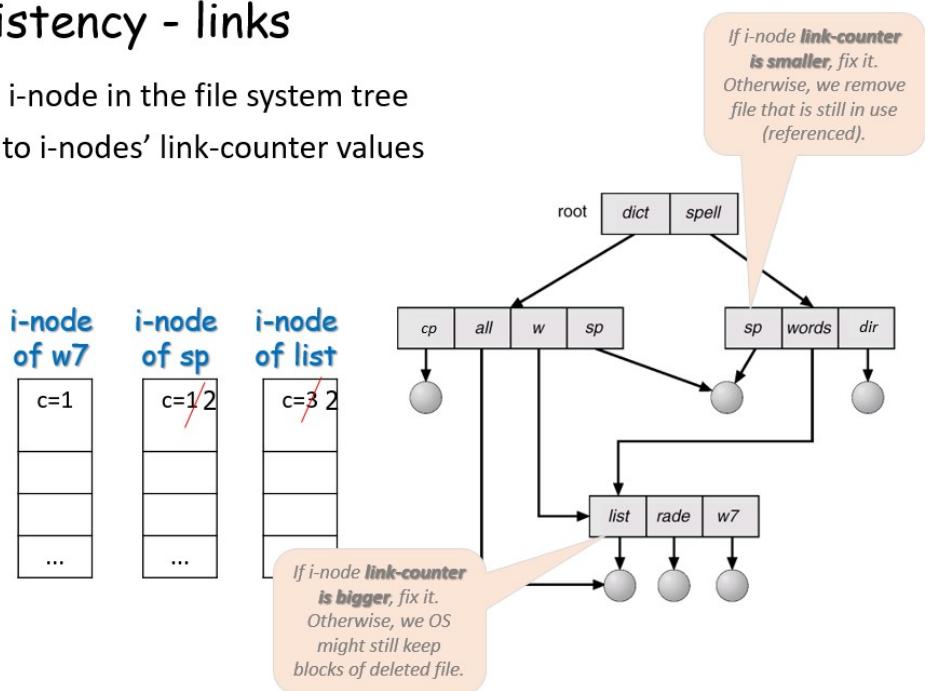
מקרה נוסף: בלוק שהוא לא בשימוש אך גם לא פנוי. זהו מקרה של בלוק חסר. במקרה זה נחליט להשאיר רק בראשית הבלוקים בשימוש.

במצב הci גרוע, הבלוק מופיע פעמיים בראשית הבלוקים התפוסים. כאן אין למערכת מושג לאיזה קובץ הבלוק שיר (אללא אם מתחזקים גיבוי). לשם כך מערכת הפעלה תשכפל את הבלוק (העתקה למעשה) ובכך שנייהם יהיו בלוקים שונים. יכול שבבלוק החדש יהיה זבל אבל אנחנו בטוחים ש-2 הקבצים לא משתמשים באותו בלוק.

File system consistency - links

- Count references to each i-node in the file system tree
- Compare these numbers to i-nodes' link-counter values

file name	number of references	i-node link-counter value
w7	1	1 ✓
sp	2	1 ✗
list	2	3 ✗
...



בדיקות עקביות על ידי בדיקה של תתי-קיימות:

בנוסף לבדיקה עקביות הבלוקים, כדי בדיקת מערכת הקבצים יכול גם לבדוק עקביות ברשומות התיקיות וכוללת ספירת הפניות לכל node-i בעץ מערכת הקבצים והשוואתן למונה הקישורים של ה-nodes-i. במקרה של שגיאה, המערכת מתקנת את המצב.

בנוסף, מערכת הקבצים בודקת את מערכת התיקיות, משתמשת במקריםים לכל קובץ כדי לעקוב אחר כמה תיקיות מכילות כל קובץ. היא משווה את המספרים הללו עם מונה הקישורים שמאוחסן ב-node-i כדי להזות שגיאות.

שני סוגים של שגיאות יכולות להתרחש: מונה הקישורים ב-node-i יכול להיות גבוהה מדי או נמוכה מדי. במקרה של שגיאה, המערכת מתקנת את המצב.

בדיקות נוספות גם אפשריות, כמו בדיקה שמספר ה-node-i לא גדול ממספר ה-nodes-i בדיסק, שמצוין שהתיקייה נפגעה.

בדוגמה שלנו לקובץ list רשום שיש 3 לינקים ב-i-node אך בפועל יש 2 רפרנסים אז מערכת הפעלה תבצע תיקון וכך יהיה זהה.

File system Reliability - Dump (a.k.a. backup)

- **Full dump**
- **Incremental dump**
- Should data be compressed before dumped?

recommended dump frequency: every month

recommended dump frequency: every day

If even small amount of compressed data is damaged, we might not be able to uncompress it, and thus loss all the backup data.

• Physical dumps

- simple, fast
- dumps also unused blocks and bad blocks
- can't skip specific directories
- can't retrieve specific files from backup, but only the whole disk

We dump disk blocks, one by one, to backup disk.

• Logical dumps

- widely used
- free blocks list not dumped – should be restored
- deal correctly with *sparse files*

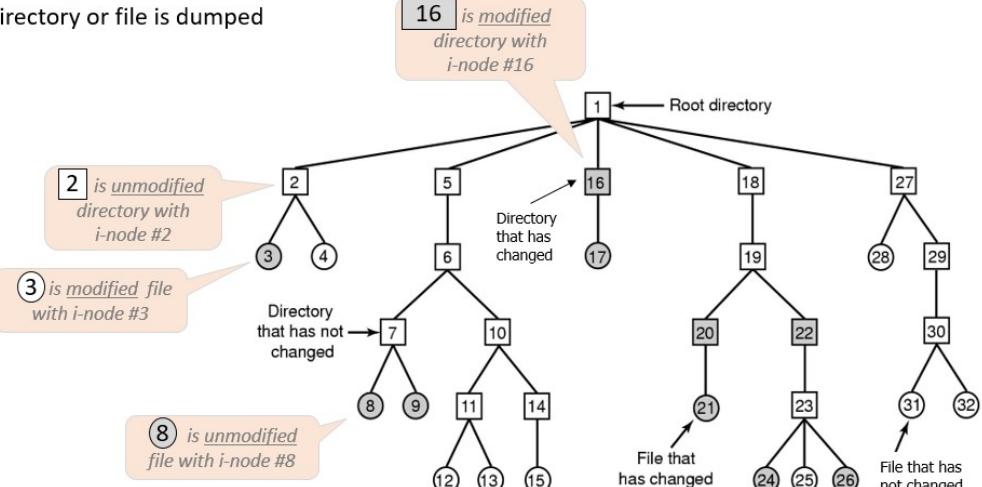
We scan directories tree, and backup each file (or only selected files) in this tree.

File System Reliability incremental dump example

- each directory and file has **dirty bit**
 - turned on when directory or file is modified
 - turned off when directory or file is dumped

directory changes when we:
 • add file
 • remove file
 • rename file
 • add directory
 • remove directory
 • rename directory
 • add hard link
 • remove hard link
 ...

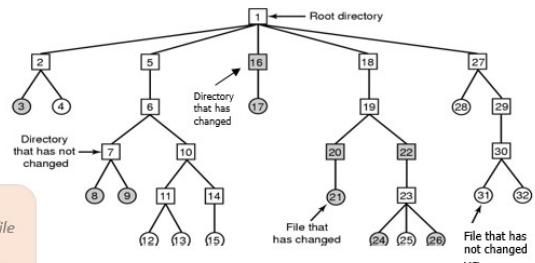
file changes when we modify its content



File System Reliability incremental dump example

- Mark modified files and all directories
- Unmark directories that do not contain marked files
- Scan bitmap in numerical order and dump all directories
- Dump files

Note: in order to restore a file, we need first to restore all directories in the path of the file, and only then to restore the file (so that we would be able to paste the file in its destination directory). So, we need first to dump all the directories, and only then to dump all the files.



Bitmap indexed by i-node number

- | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 | 32 |
|---|---|---|---|---|---|---|---|---|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
- | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 | 32 |
|---|---|---|---|---|---|---|---|---|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
- | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 | 32 |
|---|---|---|---|---|---|---|---|---|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
- | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 | 32 |
|---|---|---|---|---|---|---|---|---|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|

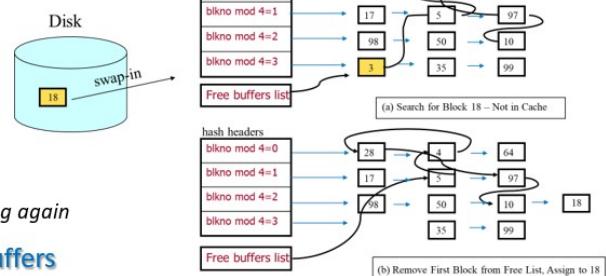
If a directory contains modified file, all the directories on the full path to this file must be backed up. We need this in order to be able to restore this file.

Thus, we backup also unmodified directories: 1, 5, 7, and so on.

Buffer Cache - Retrieval

Possible scenarios while we look for a block:

- The block is found in the hash table, and is free**
 - the buffer is marked "busy"
 - buffer is removed from free list
- The block is found in hash and it is "busy"**
 - process is BLOCKED until the buffer is freed, then starts searching again
- The block isn't found in the hash , and there are free buffers**
 - a free buffer is allocated from the free list
- The block isn't found in the hash , and in searching of the free list for a free buffers, "delayed-write" buffer is found**
 - write delayed-write buffer to disk
 - move it to the end of the list, and keep searching for a free buffer
- The block isn't found in the hash , and free list is empty**
 - allocate additional buffer in RAM for the needed block



if process is BLOCKED, when the process is re-scheduled, it should search for the required block again

איבוי מערךת הקבצים

איבוי מערךת קבצים הוא תהליך חינוי שמטרתו להגן על המידע מאובדן עקב תקלות חומרה, תוכנה או שאיגיות משתמש. ההשלכות של אובדן מערךת קבצים יכולות להיות קטסטרופליות, אפילו יותר מהרס של המחשב עצמו. בנו-אוד לצד המחשב הפיזי, שנייתן להחליפ, המידע שנאנבד ממערכת הקבצים יכול להיות בלתי ניתן לשחזור, וכן איבוי מערךת הקבצים הוא קריטי.

השימוש באיבויים נועד ל-2 מטרות עיקריות: התואשות לאסון וההתואשות מטעויות משתמש. עם זאת, תהליכי האיבוי אינם פשוט כפויו שנראה במבט ראשון, וישנן מספר אסטרטגיות לאיבוי יעיל ופקטיבי:

גיבויים חלקיים אין צורך לאבות את כל מערכת ההפעלה, אלא רק תיקיות ספציפיות והתקולה שלהן. לדוגמה, תוכנות ותיקיות זמניות אין קרייטיות מאחר וניתן להתקן מחדש תוכניות ותיקיות זמניות בכל מקרה נמחייב.

גיבויים מצטברים על מנת להשור בזמן ומקום, המערכת תבצעגיבי מלא בצורה תקופתית ואז גיבויים יומיים של קבצים שונים מאז הגיבוי האחרון. הדבר מסביר את השחזו, שכן יש לשחזר קודם את הגיבוי המלא האחרון, ולאחר מכן את כל הגיבויים התוספתיים בסדר הפוך. סכום יותר מתחכחות יכול להוריד את זמן הגיבוי ואת דרישות האחסון.

יש 2 איסות לגיבוי המידע: גיבוי פיזי וגיבוי לוגי.

גיבוי פיזי: גישה זו מעתיקה בלוקים מהדיסק בצורה סדרתית. הוא פשוט ומהיר אבל לויה בغمישות בבחירה הティקיות או קבצים ייחדים. זהו גיבוי אידיאלי עבור תוכנות גיבוי שדורשות רמת אמינות גבוהה.

גיבוי לוגי:

שיטה זו בוחרת תיקיות וקבצים ספציפיים לגיבוי, דבר הרופך את תהליך הגיבוי לרוב צדי ויעיל. זאת הgesha המועדף על ידי רוב פעולות הגיבוי ובשימוש נרחב במערכות כמו יוניקס.

בגישה זו, שחזור מערכת קבצים מהדיסק המאגובה מתחילת ביצירת קבצים ריקה על הדיסק ואז שחזור הגיבוי המלא האחרון, ולאחר מכן כל גיבוי תוספני. התהליך צריך להתמודד גם עם בעיות כמו קישורים, חורים בקבצים של AXUN, וקבצים מיוחדים שאינם צריכים להיבוט.

נפרט יותר את התהלים:

- א. כל תיקיה וקובץ מכלים סיבית שנקראת "מלוכל".
- ב. הסיבית מופעלת כאשר מתרחש שינוי בתיקיה או בקובץ.
- ג. הסיבית מתכבה כאשר מבצעים גיבוי של התיקיה או הקובץ.
- ד. שינויים בתיקיה יכולים לכלול: הוספה קובץ, הסרת קובץ, שינוי שם של קובץ, הוספת תיקיה, הסרת תיקיה, שינוי שם של תיקיה, הוספה קישור קשייח, הסרת קישור קשייח.
- ה. שינויים בקובץ מתרחשים כאשר משנים את תוכנו.

במהלך תהליך הגיבוי:

- א. מסמנים קבצים שונים ואת כל התיקיות.
- ב. מבטלים את הסימון של תיקיות שאין מכילות קבצים מסוימים.
- ג. סורקים את מפת הביטים בסדר מספרי ומבצעים גיבוי של כל התיקיות.
- ד. מבצעים גיבוי של כל הקבצים.

חשוב לציין: כדי לשחזר קובץ, יש לשחזר קודם את כל התיקיות שבנתיב של הקובץ, ורק לאחר מכן לשחזר את הקובץ עצמו. זה אפשר לנו להדביק את הקובץ בתיקיה היעד. לכן, חיבאים לבצע קודם גיבוי של כל התיקיות, ורק לאחר מכן גיבוי של כל הקבצים.

אם תיקיה מכילה קובץ שונה, כל התיקיות שבנתיב המלא לקובץ זה חייבות להיבוט. זה נדרש כדי לאפשר שחזור של הקובץ.

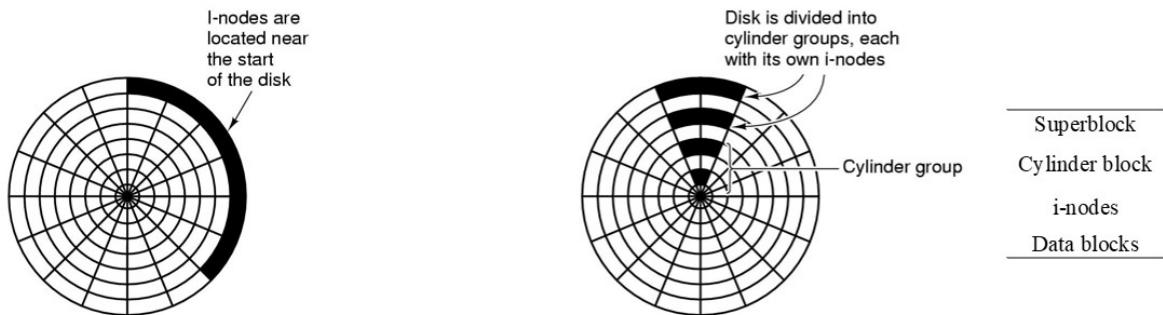
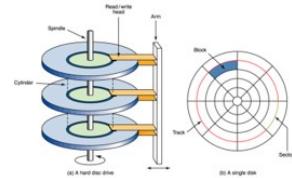
לכן, במהלך תהליך האיבוי, אנו מגבאים אם תיקיות שלא שונו, כמו: 1, 5, 6, 7, וכו'.

File systems: outline

- ❑ Concepts
- ❑ File system implementation
 - Disk space management
 - Reliability
 - Performance issues
- ❑ NTFS
- ❑ NFS

Performance: Reducing disk arm motion

- Block allocation
 - assign consecutive blocks on same track
 - possibly rearrange disk periodically
- Where should i-nodes be placed ?
 - Start of disk – $\frac{1}{2}$ rotation in average from i-node to first data block of file
 - Middle of disk – $\frac{1}{4}$ rotation in average
 - divide disk into cylinders and place i-nodes blocks and appropriate files blocks on same cylinder



Unix performance enhancements

- **two block sizes are supported** – bigger files get bigger block size for faster file data access
- **caching of (fileName, i-node) pairs**

❑ buffer cache

- cache of recently used disk blocks, in RAM buffers (buffer per disk block)
- this cache is **stored in OS address space**
- if OS finds a disk block requested by process in some buffer in the cache, OS **copies this block** to the process address space

❑ delay-write

- dirty disk blocks may be written to buffer cache before they are written to disk

סוגיות ביצועים

גישת לדיסק היא בהרבה מונחים יותר איטית מאשר גישה לזכרון. אם קריאה של מילת זיכרון יכולה לקחת 10 ננו-שניות, קריאה מהdisk הקשייה יכולה להתבצע בקצב של 100 מגה-ביט לשניה. בנוסף, יש לקחת בחשבון את הזמן הנדרש לאיתור המסלילה המתאימה והמתנה לסקטור הנכון להגיא מתחם לראש הקריאה.

ביצועים: הפחתת תנודת זרוע הדיסק

מדד קריטי לשיפור ביצועי מערכת הקבצים בדיסק קשייה קלאסי הוא התנועה של זרוע הדיסק, או במלחים אחרות, זמן האיתור. ניתן לעשות זאת בעזרת מספר טכניקות:

שיבוץ בצורה מעגלית

טכניתה נוספת כוללת התחשבות בכך שהדיסק הוא מעגלי. כאשר יש הקצאות בлокים, המערכת שואפת למקום בлокים סמוכים בקובץ בתוך הצילינדר. הדבר מפחית את הדרישה מהדיסק לאתר בлокים במיקומים שונים בциינדר, דבר המסייע ביצועים.

איפה צריך למקם את ה-nodes-i?

במערכות קבצים אשר משתמשות ב-inode, קריאת קובץ קטן לעתים דורשת 2 גישות לדיסק. הגישה הראשונה היא עבורי הבאת ה-node*i* עצמו ועוד אחד בשביל בлок המידע. שיבוץ לא יעיל של ה-inode*i* יכול להגדיל את זמן האיתור. לשם כך ישנו מספר אסטרטגיות לשיבוץ ה-nodes-i בדיסק:

- א. בתחילת הדיסק – במשמעותו, חצי סיבוב מה-node*i* לblk הנתוני הראשון של הקובץ.
- ב. באמצע הדיסק – במשמעותו, רביע סיבוב.
- ג. מחולקים את הדיסק למעגלים וממנים את בлокי ה-nodes-i ובלוקי הנתונים המתאימים על אותו המעגל.

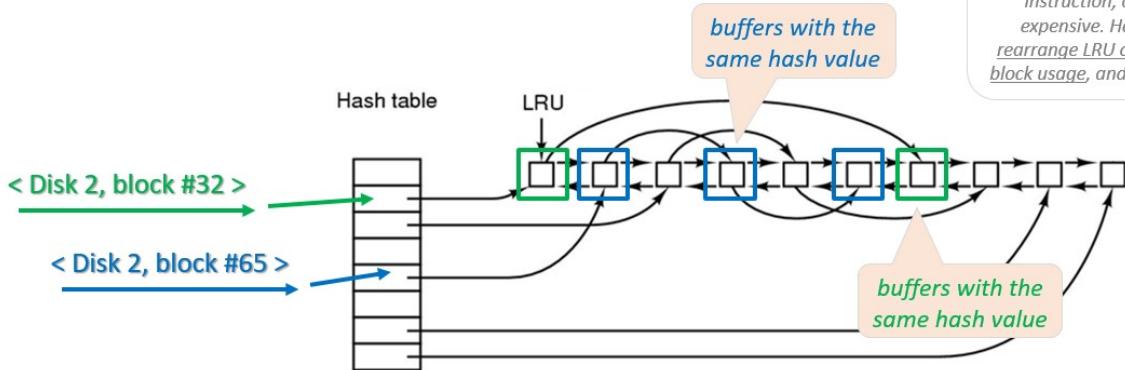
Unix Buffer cache replacement

- Each cache buffer has **<Disk id, block #> header**
- Buffers are on a doubly-linked list in LRU-style order
 - used for **buffer eviction**
- Buffers are inserted into Hash table, by hash(<disk id, block #>)
- used for **fast lookup**

There may be different File Systems on different disks. OS should support working with all of them.

When we studied Page Replacement algorithms, we said that LRU is not applicable due to its run time complexity. Why LRU is applicable here?

We need to rearrange LRU order of pages several times for a single instruction, and this is very expensive. Here, we need to rearrange LRU order for each disk block usage, and this is **acceptable**.



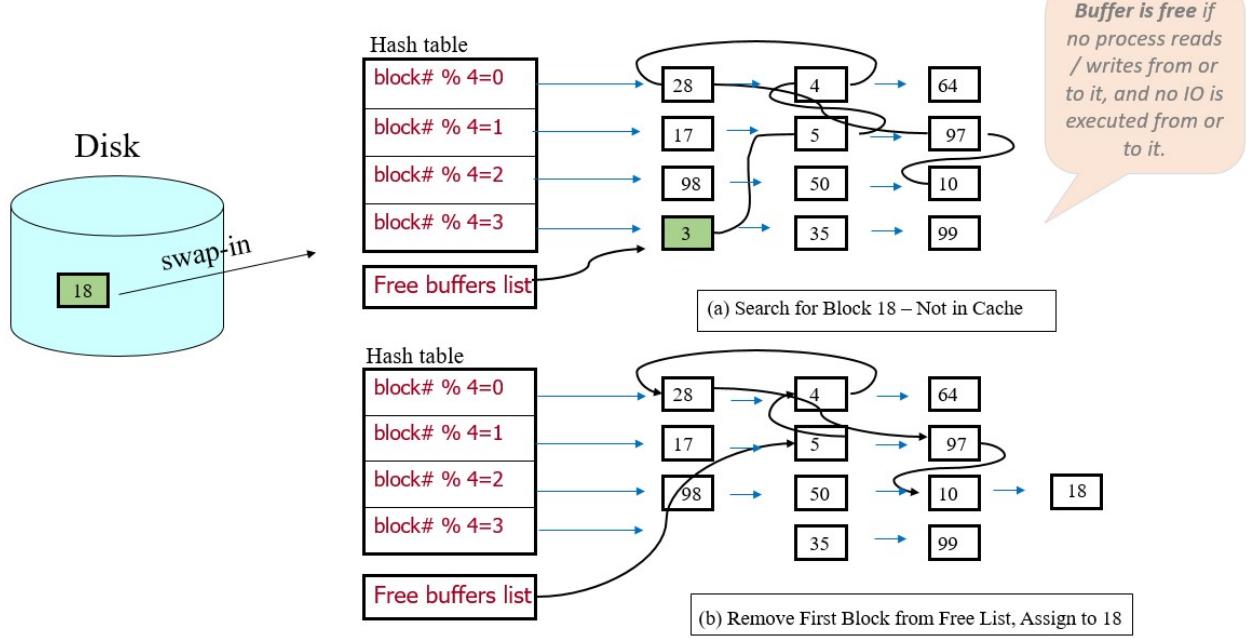
Why not pure LRU?

- Some **blocks are critical** and should be written as quickly as possible (for example, *i-nodes*)
 - Insert critical blocks at the head of the LRU list, to be replaced (evicted) soon and thus written to disk
- Some blocks are likely to be **used again (directory blocks)**
 - Insert such blocks to the end of the LRU list
- **Partly filled blocks** being written go to the end to stay longer in the cache

some Operating Systems may not use delay-write, like MSDOS

there is system **daemon thread** that calls sync every 30 seconds, and **write dirty buffers** to disk, so that we would not loss files' changes in a case of power failure

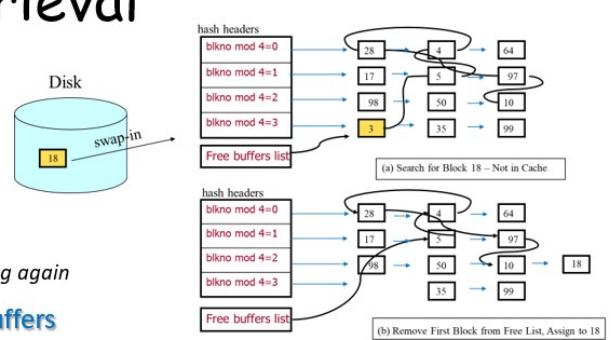
Scenarios for Retrieval of a Buffer



Buffer Cache - Retrieval

Possible scenarios while we look for a block:

1. **The block is found in the hash table, and is free**
 - I. the buffer is marked "busy"
 - II. buffer is removed from free list
2. **The block is found in hash and it is "busy"**
 - process is BLOCKED until the buffer is freed, then starts searching again
3. **The block isn't found in the hash , and there are free buffers**
 - a free buffer is allocated from the free list
4. **The block isn't found in the hash , and in searching of the free list for a free buffers, "delayed-write" buffer is found**
 - write delayed-write buffer to disk
 - move it to the end of the list, and keep searching for a free buffer
5. **The block isn't found in the hash , and free list is empty**
 - allocate additional buffer in RAM for the needed block

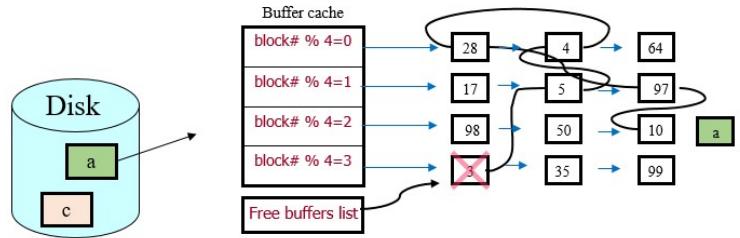


if process is BLOCKED , when the process is re-scheduled, it should search for the required block again

Fine points of Buffer-cache block retrieval

process A

```
FILE* out = fopen("output.txt", "w+");  
char c = fgetc(out);  
fclose(out);
```



- process **A** tries to access block 'a'
- block a is not found in the Buffer cache
- => OS blocks **A**, finds free buffer, locks this buffer, and initiates IO of a from Disk
- process **B** also tries to access block 'a', and finds it locked (busy) => **B** is blocked
- after block a is loaded, and fgetc() is finished, **A** and **B** are unblocked
- **B** must search again for 'a', since:
 - the buffer containing 'a' might be busy again
 - the buffer might be reused for loading some other block 'c', say, by some process **C**
 - the buffer might be reused, but maybe block 'a' is loaded to some other buffer, say, by the process **A**

הטמנת בЛОקים

טכנית זו משלחת תפקוד חשוב בשיפור ביצועי מערכת הקבצים על ידי הפחתת הדרישה לגישה לדיסק. לפי גישה זו, בלוקים אשר יש בהם שימוש תDIR יישמרו בזיכרון, כאשר המטען הוא למעשה אוסף של בלוקים אשר אמורים להיות בדיסק אך נשמרים בזיכרון על מנת לשפר את הביצועים.

ישנו 6 נקודות מפתח אשר באים לידי ביטוי במימוש מנגנון זה:

אלגוריתמי ניהול המטען:

ישנו מספר אלגוריתם אשר ניתן להשתמש בהם כדי לנהל את המטען, אך השכיח ביניהם כולל בדיקת בקשת הקריאה על מנת לבדוק האם הבלוק המבוקש כבר נמצא במעון. אם כן, ניתן לטבל בבקשת הקריאה מבלי לאחסן לדיסק. אחרת, יש צורך לקרוא את הבלוק למעון ואז העתקתו למקום הדרוש. בצורה זו, ניתן לבצע פניות חוזרות ונשנות לאותו הבלוק מתוך המטען.

שימוש באש לחיפוש מהיר:

לעתים קרובות המטען ניכלו הרבה בלוקים, ולכן נדרשות שיטות יעילות כדי לקבוע האם הבלוק נמצא או לא. דרך נפוצה כוללת שימוש באש עבור מזהה הדיסק ומספר הבלוק כדי לחפש את התוצאות בטבלת האש. בלוקים בעלי אותו ערך האש ישרשו ביחד ברשימה דואית כיוונית.

אסטרטגיית החלפת באפרים במעון:

כאשר מעלים בלוק למעון מלא, בלוק אחד חייב להימחק, ואף להיכתב מחדש בדיסק אם עבר שינוי. לשם כך יש צורך באסטרטגיה להחלפת באפרים במעון. האסטרטגיות הללו דומות לאלגוריתמי החלפת דפים שראינו בפרק הקודם ונitin להשתמש בהם לניהוליעיל של המטען.

אלגוריתמים אלו כוללים את FIFO ואת אלגוריתם ההזדמנויות השנייה, אך המתאימה ביותר לכך היא אלגוריתם ה- LRU. nim לב שבמקרה הזה אין את הבעיות במעון כפי שהיא במנגן החלפת דפים, משום שהצורך בסידור

מחודש של סדר הבאים בא ידי ביטוי רק כאשר מתבצע שימוש בבלוק, ולא עבר כל הוראה.

סכמת LRU מותאמת

במקרה שלנו, שימוש באלגוריתם LRU המדוקן אינו תמיד יdeal. בעוד שnitן לקרוא בלוקים קרייטיים, כמו nodes, לתוך המטמון, ברגע שבוצע בהם שינוי חובה לכתוב אותם בצורה מיידית לדיסק. הסיבה לכך היא שבמקרה של קרייסט המערכת שאינה כותבת בלוקים קרייטיים ישירות לדיסק, המערכת עלולה להישאר במצב לא יציב. לשם כך האלגוריתם המותאם ייקח בחשבון 2 דברים: הסיכוי שיש צורך בבלוק שוב והחשיבות של הבלוק לעקבות מערכת הקבצים.

הדבר בא לידי ביטוי בשתי דרכים אפשריות. הבלוקים הקרייטיים, כמו nodes, מוכנסים בראש הרשימה של LRU, כדי שייהו הראשונים להימחק מהמטמון ולהיכתב לדיסק. בלוקים שסביר להניח שיישונמו שוב, כמו בלוקי תיקיות, מוכנסים בסוף הרשימה של LRU. בלוקים שבוצע בהם שינוי ואינם מלאים מועברים בסוף כדי להישאר יותר זמן במטמון.

במקרה שבו בלוק לא נמצא במטמון, מערכת ההפעלה חוסמת את התהיליך המבוקש, מוצאת באפר פנו, נועלת אותו, ומתחילה לטען את הבלוק מהדיסק. אם תהיליך אחר מנסה לגשת באותו בלוק במהלך הטעינה, הוא נכנס למצב של block עד שהטעינה מסתיימת.

מערכת ההפעלה של אונח משתמש גם בקריאה המערכת sync, שמכריחה בלוקים שעברו שינוי להיכתב לדיסק באינטראולים קבועים. זה מבטיח שבמקרה של קרישה, המידע שנמצא במטמון לא יאבוט.

המטמון הוא מטמון של בלוקים של דיסקים, ולא מטמון של דפים של זיכרון.

תרחישים לאחזר באפר

נבהיר כי באפר הוא פנו אם אין תהיליך שקורא (כותב) ממנו או אליו ואין אף פעולה O המבוצע ממנו או אליו. יתרכנו מספר תרחישים בעת החיפוש אחר בלוק:

א. הבלוק נמצא בטלת hash, והוא פנו - זהו התרחיש הטוב ביותר, הבאפר מסומן כ-"עסוק" והוא מסור מרישימת הבלוקים הפנוים.

ב.. הבלוק נמצא ב-hash והוא "עסוק" - ככלומר הבלוק בשימוש על ידי תהיליך אחר, אז התהיליך יכנס למצב של block עד שהבאפר יתפנה ואז הוא יחפש שוב.

ג.. הבלוק לא נמצא ב-hash, ויש באפרים פנוים, אז מערכת הפעלה פשוט תקצת באפר מרישימת הבאים הפנוים.

ד. הבלוק לא נמצא ב-hash, ובחיפוש אחר באפרים פנוים, נמצא באפר "כתיבה מעוכבת", אז מערכת הפעלה תבצע את הכתיבה ואז תזיז אותו בסוף הרשימה ותמשיך לחפש אחר באפרים פנוים.

ה. הבלוק לא נמצא ב-hash, ורישימת הבאים הפנוים ריקה - במקרה הגורע ביותר - יש צורך להקצות באפר נוסף לבlok.

דוגמת ריצה

א. תהיליך A מנסה לגשת לבlok 'a'.

ב. בלוק a לא נמצא במטמון הבאפר, כתוצאה לכך מערכת ההפעלה חוסמת את A, מוצאת באפר פנו, נועלת את הבאפר הזה, ומתחילה O של a מהדיסק.

ג. תהיליך B גם מנסה לגשת לבlok 'a', ומוצא אותו נעלם (עסוק) => B מוחסם.

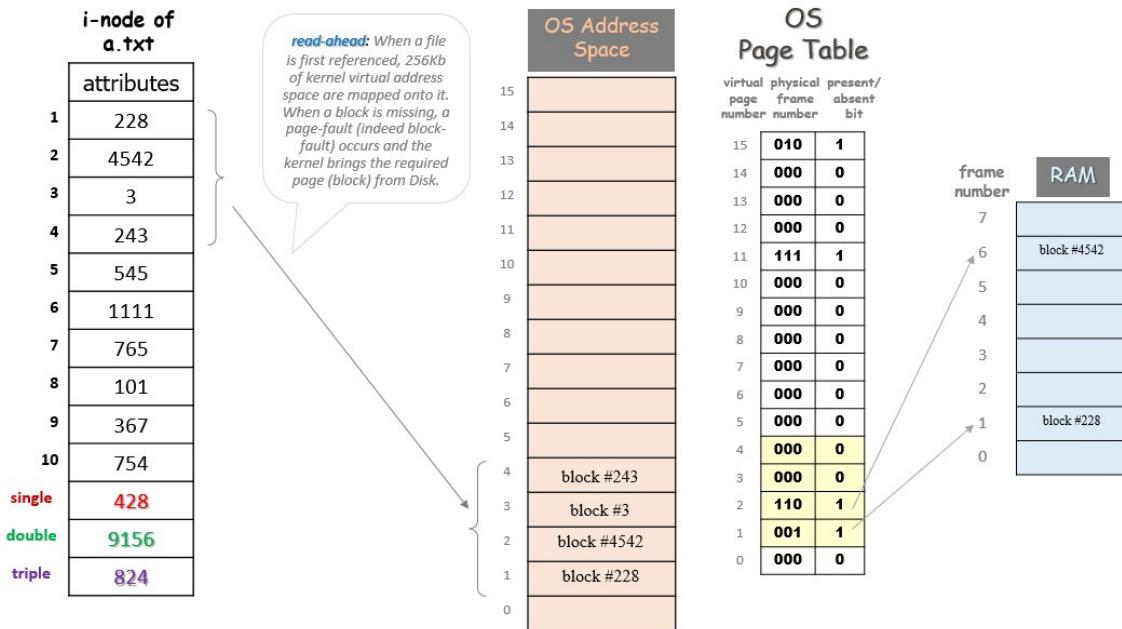
ד. לאחר שהבלוק a נטען, וgetc() מסתים, A ו-B משוחרים.

ה. B. חיב לחש שוב את 'a', לאחר ש:

ו. הבאפר שמכיל את 'a' יכול להיות עסוק שוב.

ז. הבאפר יכול להשתמש מחדש, אך אולי בлок 'a' נטען לבאפר אחר, למשל, על ידי התחילה

Caching in Windows 2000



Caching in Windows 2000

- buffer key in hash is **virtual block <file name, offset>** and not <device, physical block> (like UNIX does)
- read-ahead**: when a file is first referenced, 256Kb of kernel virtual address space are mapped onto it. When a block is missing, a page-fault (indeed block-fault) occurs and the kernel brings the required page (block) from Disk.
- Memory manager** can trade-off buffer cache size dynamically
 - more user processes → less cache blocks in RAM
 - more files activity → more cache blocks in RAM

Block Read Ahead

זאת גישה טיפה שונה אשר משתמשת בرعיוון של שימוש במתਮן מכיוון אחר. בגישה זו, מערכת הפעלה מעלה בזיכרון יזומה בЛОקים למטרון לפני שיש בהם באמת צורך - דבר המגדיל את יחס הפגיעה במתמן. הטכניקה עלייה במיוחד עבור קרייה בצורה רציפה של קבצים.

השיטה עובדת بصورة הבא:

- א. כאשר מערכת הפעלה נדרשת להביא בлок מסוים מטור קובץ, היא מבצעת את הבקשה.
- ב. לאחר ביצוע הבקשה, מערכת הקבציםבודקת מהר האם הבלוק הבא אחרי בדיסק כבר נמצא במטמון.
- ג. אם הוא לא נמצא במטמון, מערכת הפעלה מתחזמת פעלות קריאה של הבלוק הסמור, בczyfיה שהיא בו שימוש נוסף.

היתרון של גישה זו הוא שכאשר תהליך רוצה לגשת לבלוק שאינו בזיכרון, מתרחשת תקלת דף, ואז מערכת הפעלה מביאה בלוקים נוספים באמצעות מגנון החלפת הדפים כדי לפתור בעיה דומה.

חסרון בולט בגיןה זו היא עיליה רק עבור קבצים שבאמת מתבצעת בהם קריאה בצורה רציפה. אם יש צפי בגיןה אקרואית לקובץ, גישה זו הלכה למעשה רק תפגע בביצועים, שכן אין הכרח שהתוכניות יעשו שימוש בבלוקים סמוכים.

זאת ועוד, אם הבלוקים אשר יוצאים מהבאפר עברו שינוי, הם יצרכו את כל רוחב הפס של הדיסק כתיבתם בחזרה.

אסטרטגיה זו מומשאה ב-2000 Windows. נזכר כי מערכת הפעלה היא אם תהליך עם מרחב כתובות וירטואלי ותמונה תהליך משלה. כאשר המערכת הייתה פותחת קובץ, היא הייתה מוסיפה 64 דפים למרחב הכתובות שלו למטרת מיפוי הקובץ. כאשר תהליך רוצה לגשת לבלוק הראשון בקובץ, הוא היה בדף אחד למרחב הכתובות של Windows.

בעוד שהגישה הקודמת מתייחסת לנוטונים כמספר דיסק ומספר בלוק (שמתווגם בסופו של דבר למיקום בדיסק), גישה זו משתמשת בשם הקובץ וב-offset.

כדי לקבוע האם להכיל מדיניות זו, מערכת הקבצים יכולה לעקוב אחר תבנית השימוש של כל קובץ פתוח. לדוגמה, ניתן להשתמש בבית השיר לכל קובץ שמצוין האם הוא במצב של גישה סדרתית או במצב של גישה רנדומית. עם זאת, כאשר מתבצעת פעולה seek, שמעידה על גישה לא רציפה, הבית מתכבה. אם הקריאה הסדרתית ממשיכה, הבית ידלק שוב.

ב-2000 Windows, כאשר קובץ מופנה לראשונה, 256Kb של מרחב הכתובות הוירטואלי של הליבה ממופים עליו. כאשר בלוק חסר, מתרחשת תקלת דף (בעצם תקלת בלוק) והליבה מביאה את הדף (הблוק) הנדרש מהדיסק. המפתח של המטען בגישה הוא הבלוק הוירטואלי; שם הקובץ, offset ולא התקן, בלוק פיזי (כמו ש-XINU עושים). ניהול הזיכרון יכול להחליף באופן דינמי את אודל מטען המטען. יותר תהליכי של משתמשים מוביילים

לפחות בלוקים של מטמון ב-RAM, ועוד פועלות קבצים מוביילה ליותר בלוקים של מטמון ב-RAM.

File systems: outline

- Concepts
- File system implementation
 - Disk space management
 - Reliability
 - Performance issues
- NTFS
- NFS

מערכת הקבצים NTFS

NTFS - NT File System

like UNIX i-nodes Table

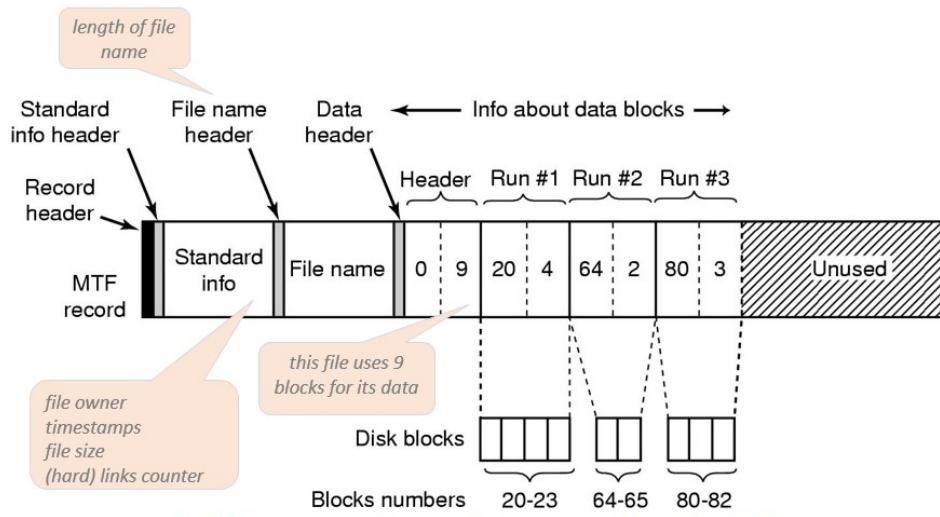
- ❑ **MFT (Master File Table)** - a table that has one or more records per file/directory
- ❑ MFT is kept in a file, in any place on disk in 2 places
 - maximal MFS size - 2^{48} records
- ❑ record contains file attributes and list of file block numbers
- ❑ **immediate file**: very small files, so that the data can be kept *directly* in the MFT record
- ❑ large files need more than one MFT record for their list of blocks - records are extended by pointing to other records
- ❑ disk blocks are assigned in *runs* (*i.e. sequential blocks on disk*), and kept as a sequence of 2 64-bit numbers – *(offset, length)*
- ❑ no limit on file size
- ❑ An attribute that is stored within record is called **resident**
- ❑ **sparse file**: file with some holes - skipped blocks (which should be zeroed). Each sequence till hole is described by separate MFT record.

for example: write first 5 blocks, than seek to 60'th block (hole) and write 3 blocks, then seek to 200'th block (hole), and so on...

in NTFS, block number is 64-bit number

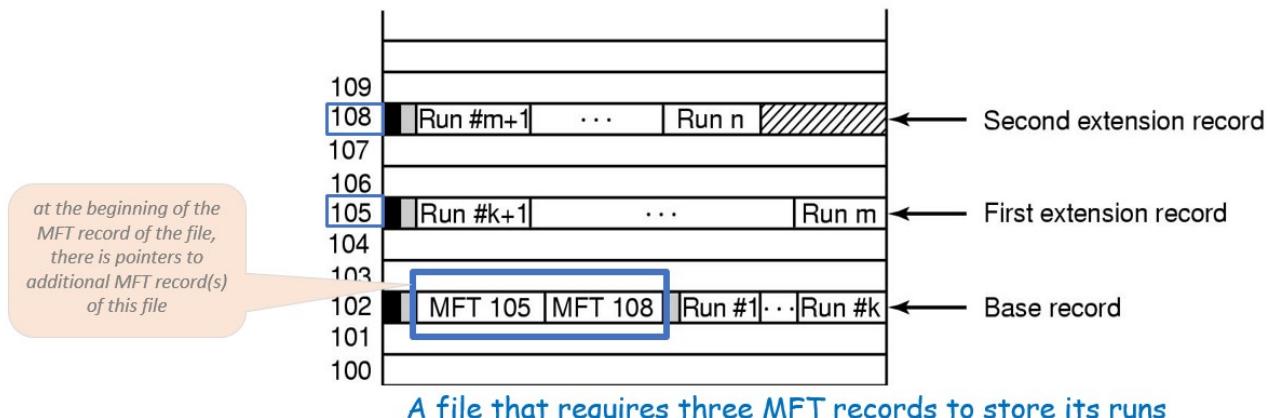
MFT record of a non-immediate file

1 Kb fixed-size record



An MFT record for a three-run, nine-block file

The MFT record of long files

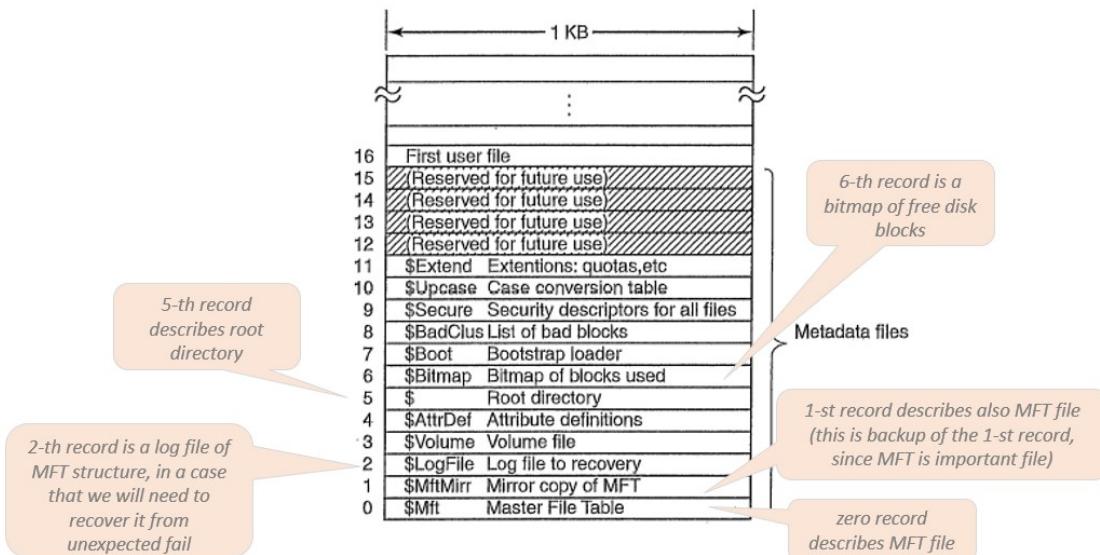


Can it be that a short file uses more MFT records than a longer file?

Yes, if small file is fragmented and longer file blocks are sequential on disk.

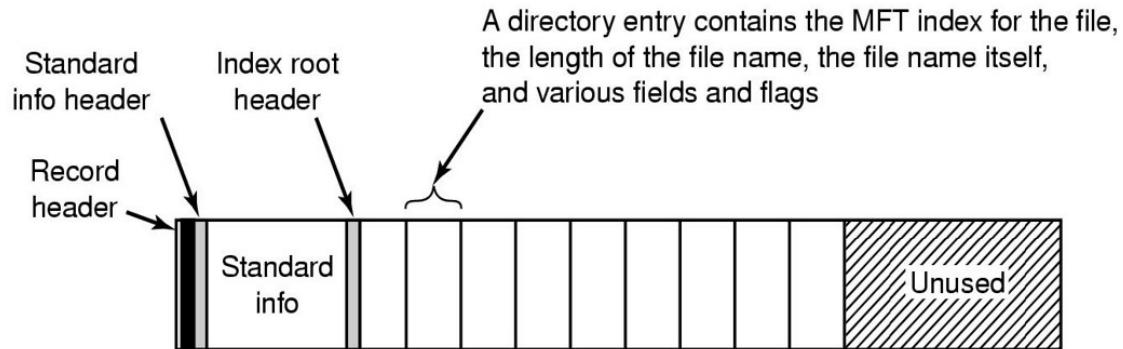
MFT metadata files

- first 16 records in MFT describe the file system
- boot sector contains MFT address

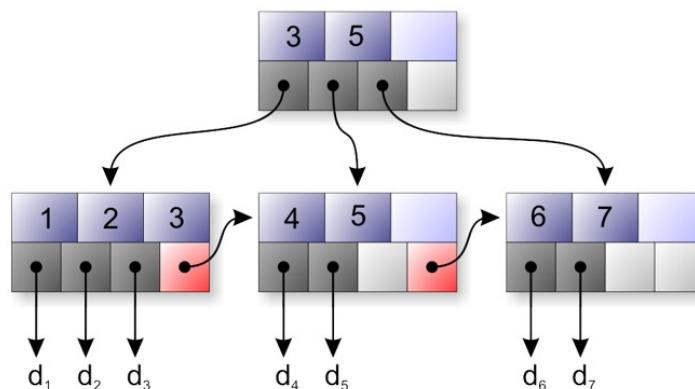


NTFS - Small directories

sequential search for needed file in the directory



NTFS - Large directories

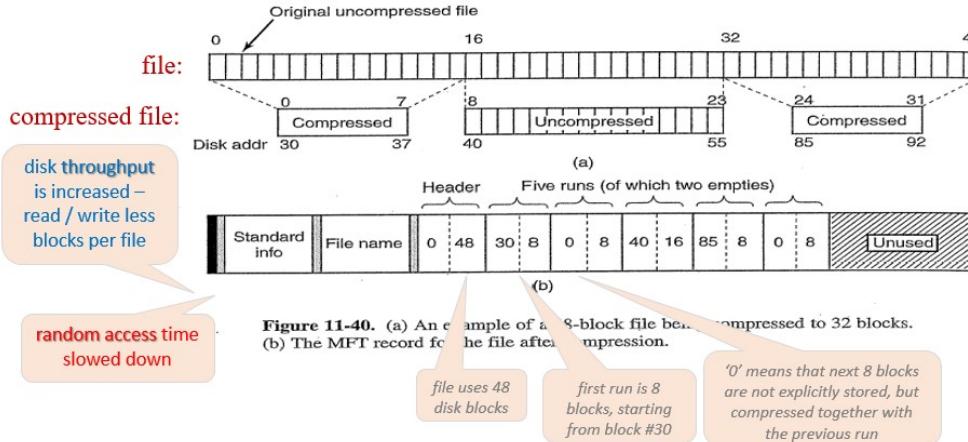


Large directories are organized as B+ trees

NTFS file compression

- Supports transparent (i.e., automatic) file compression
- Compresses (or not) in groups of 16 blocks
 - If at least one block saved (we got at most 15 compressed blocks) – writes compressed data, otherwise writes uncompressed data
- Compression algorithm: a variant of LZ77 (Lempel-Ziv)

We do not compress the whole file since we do not want to uncompress the whole file while read / write one block.v



Windows MFTs vs. Unix i-nodes

Windows MFT table	Unix i-nodes table
MFT record size is 1Kb	i-node entry size is 64-byte
MFT file may be located anywhere on disk	i-nodes table is located immediately after superblock
Name of file is written in MFT record	i-node does not contain file names
for small files, file data is in MFT record	file data is never in i-node
disk blocks are allocated by runs – fast search of sequentially allocated data blocks	disk blocks are allocated by tree of indexes – fast search of random data block

מערכת הקבצים NTFS, שנוצרה עבור Windows NT והפכה לברירת המחדל מאז XP, מציעה אבטחה משופרת, פונקציונליות מרובה, ותמכה בגודלים גדולים של מחיצות דיסק ושמות קבצים.

מערכת הקבצים תומכת במאפייני קובץ מרובים, אשר מיוצגים על ידי סטרים של בתים. היא בנויה במבנה היררכי הדומה לינוקס, כאשר שמות הרכיבים מופרדים על ידי קוים אחרים. בנוסף היא תומכת בקישורים סימוביילים וקישורים קשייחים.

מבנה מערכת הקבצים

מבנה מערכת הקבצים הוא למעשה מבנה נתונים אחד, טבלת ה-MFT או בשמו המלא: Master File Table. הרעיון מאחורי המימוש היה לאפשר גמישות מרבית בייצוג האלמנטים במערכת הקבצים. ה-MFT הוא בעצם קובץ אשר ניתן לשימוש בכל מקום במחיצת ה-MFT אין גודל קבוע והוא יכול לגודל דינמי עד לגודל של 248 רשומות.

הטבלה עצמה מכילה רשומות, כאשר גודל כל רשומה הוא KB1. כל רשומה מייצגת קובץ או תיקייה וכוללת מאפיינים כמו שם, חתימות זמן וכתובות בלוקים. עבור קבצים אדוליפ, ניתן להשתמש במספר רשומות MFT, כאשר הראשונה מחזיקה מצביעים לרשותן נוספת.

לניהול רשומות MFT ריקות, יש שימוש ב-[bimap](#).

מבנה רשומה MFT

כל רשומה מחזיקה סדרה של זוגות המtauרים מבנה תוכנה וערך. מבנה התוכנה מצין את סוג התוכנה, אורכה ואת מקום הערך שלה. התוכנות יכולות להיות בתוך רשומה ה-MFT או מחוץ לרשימה במקום אחר בדיסק.

קבצי מטאדתאטה

16 רשומות ה-MFT הראשונות מוקצחות עבור קבצים מיוחדים, כאשר כל קובץ מתחילה בסמן הדולר. הקבצים הללו כוללים מידע על קובץ ה-MFT, קובץ יומן, תיקיית השורש ועוד.

תכונות

מערכת הקבצים מגדרה 13 תכונות אשר יכולות להופיע ברשותן ה-MFT, כולל מידע בסיסי, שם הקובץ, רשימת תכונות ועוד.

הकצאות אחסון (מבנה הרשותה)

בשביל הייעול, מערכת הקבצים מקצת בלוקים בראיצות של בלוקים במקומות רציפים (במילים אחרות, רציפים או ריצות) ככל הניתן. לעומת זאת, בלוק לוגי של קובץ נמצא בבלוק פיזי כלשהו בדיסק, אז המערכת תנסה למוקם את הבלוק הלוגי הבא בבלוק הפיזי הסמוך. גישה זו נועדה לצמצם את הפרגמנטציה הפנימית.

בלוקים בתוך הקובץ מתוארים על ידי סדרה של רשומות. כל רשומה מגדרה את הסדרה של הבלוקים הרציפים בקובץ. עבור קבצים ללא חורים (קבצים קטנים), יש רק רשומה אחת. לקבצים עם חורים, יש שימוש במספר רשומות לתיאור בלוקים במקומות שאינם סמוכים.

בנוסף, NTFS תומכת בקבצים חסרים (sparse files) שהם קבצים עם חורים - בלוקים שנDELgo (אשר צריכים להיות מאופסים). כל רציף עד החור מתואר על ידי רשומה MFT נפרדת. כאשר מתבצעת קריאה של קובץ כזה, המערכת מתייחסת למידע החסר על אפסים.

כל רשומה מתחילה במבנה המציין את היחס של הבלוק הראשון ברצף ואת היחס של הבלוק הראשון שאינו מכוסה על ידי הרשותה. בצד השני של המבנה ישנו זוגות של כתובות דיסק ואורכי הרצפה, על מנת לציין היכן הבלוקים נמצאים וכמה יש מהם ברצף אחד.

כאמור לעיל, רשומות ה-MFT הן גמיישות. אין חסם עליון לגודל הקבצים אשר ניתן ליצא בצוורה זו. זוג יחיד יכול לבטא מספר רב של בלוקים סמוכים, ועל מנת לצמצם את הגודל של הזוגות הללו, ישנו שימוש בטכניקות כיווץ.

מערכת הקבצים מאפשרת לקבצים גדולים או קבצים עם הרבה מקטעים (הרבה רציפים) להשתמש במספר רשומות, על ידי אחסון אינדקסים לרשותן ההרחבה ברשותן הבסיס.

תיקיות במבנה הקבצים מיוצגות גם הן כרשומות MFT. תיקיות קטנות מאוחסנות בצורה לינארית עם מבנה בגודל סופי ותיקיות גדולות משתמשות בעציהם+B+עבור חיפוש והכנסה עילימ.

מערכת ה-NTFS תומכת בדוחיסת קבצים שקופה, דבר המאפשר לקובץ להיווצר במצב דחוס, היכן שדוחיסה ושחרור מתבצעים אוטומטיות מבלתי שתהליכים מודעים לכך.

כאשר בוצעה כתיבה לקובץ אשר סומן לדוחיסה, NTFS בוחנת את 16 הבלוקים הלוגיים הראשונים בקובץ. במידה הצורך, היא מרים אлогריתם דוחיסה על בלוקים אלו. אם ניתן לאכלס את תוצאת הדוחיסה בכל הבלוקים 15 והם נכתבים לדיסק במצב מכועז. במצב האידיאלי התהילה מתבצעת בሪיצה אחת.

אם המידע המכועז עדיין נדרש 16 בלוקים, אז 16 הבלוקים יוכתבו במצב לא דחוס ומערכת הקבצים תבחן את 16 הבלוקים הבאים ותנסה לבצע עליהם את התהילה זה.

החלקים הדחוסים והחלקים הלא דחוסים מאוחסנים במספר רצפים, כאשר כל אחד מהם נמצא ברשותת ה-MFT של הקובץ. כדי לקרוא קבצים דחוסים, מערכת הקבצים בוחנת אלו רצפים מכועזים בהתבסס על כתובות הדיסק, כאשר כתובת 0 מצינית חלק סופי של 16 בלוקים מכועזים.

גישה אקראית לקבצים מכועזים היא אפשרית, אבל יחסית יקרה, כיוון והוא עלולה לדרוש קריאה מרובה (ושחרור רב) של ריצות מכועזות רבות. מכאן, השימוש ביחידת הדוחיסה של מערכת הקבצים הוא עדיף עבור קבצים שאין בהם גישה אקראית באופן תדרי.

בשילובה למערכת הקבצים של יוניקס, היא מציעה גמישות רבה יותר בניהול הקבצים. לדוגמה, גודל הרשומה ב-MFT הוא 1KB, בעוד שב-Unix הוא 64 בתים בלבד. בנוסף, שמות הקבצים מאוחסנים ברשותת MFT, בעוד שב-Unix הם לא מאוחסנים ב-eodeh-. עבור קבצים קטנים, המידע של הקובץ נמצא ישירות ברשותת MFT, בעוד שב-Unix המידע של הקובץ לעולם לא נמצא ב-eodeh-. בנוסף, בלוקים של הדיסק מוקצים בritchot ב-NTFS, מה שמאפשר חיפוש מהיר של בלוקים שהוקצו בצורה רציפה, בעוד שב-Unix הבלוקים מוקצים בעז של אינדקסים, מה שמאפשר חיפוש מהיר של בלוק נתונים אקראי.

File systems: outline

- Concepts

- File system implementation

- Disk space management
- Reliability
- Performance issues

- NTFS

- NFS

Distributed File Systems (DFS)

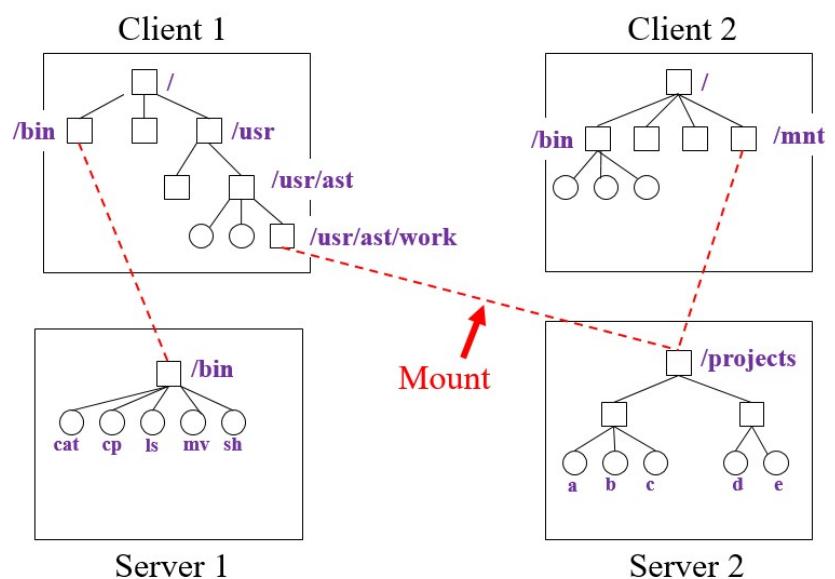
מערכת קבצים מסווג DFS הן מכונות מחוברות שאינן מושתפות זיכרון. דוגמה אחת ל-DFS היא NFS, שהיא אוסף של שירותי ולקוחות שיכולים לפעול על מכונות שונות עם מערכות הפעלה שונות. במערכת זו, כל מכונה יכולה לפעול גם כלקוח וגם כשרת.

סקירה של NFS

מערכת קבצים הרשת NFS של SAN מיקראוסטטם מאפשרת למחשבים שונים לחלק מערכת קבצים משותפת, באמצעות שירותי ולקוחות. אפשר להפעיל את NFS על רשת מקומית או רשת רחבה עם שרת מרוחק מהלקוח. כדי לדון בנFS, נדרש לדון בארכיטקטורה של NFS, בפרוטוקול ובמימוש.

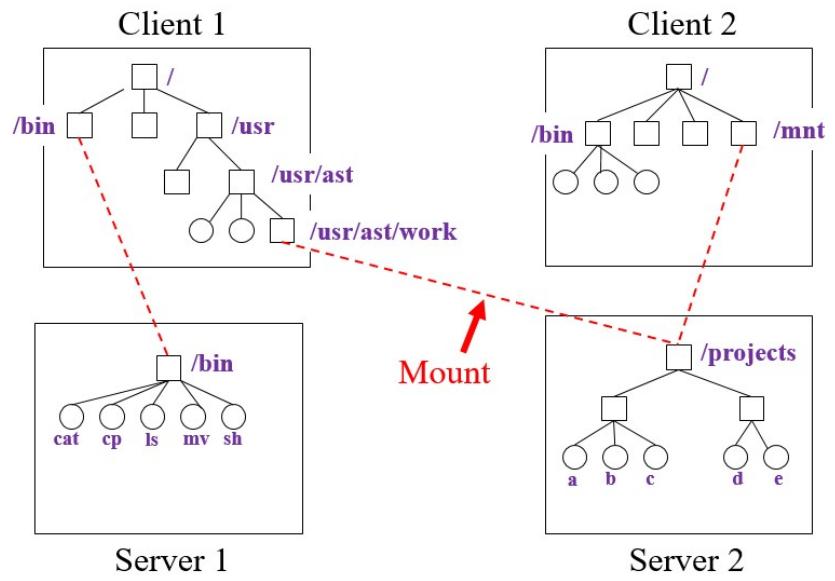
DFS - Distributed File Systems

- DFS** is a collection of interconnected machines that do not share memory
- NFS** is a collection of interconnected servers and clients - can be different machines with different OSs
- Any machine can be both client and server
- Clients access directories on servers by mounting



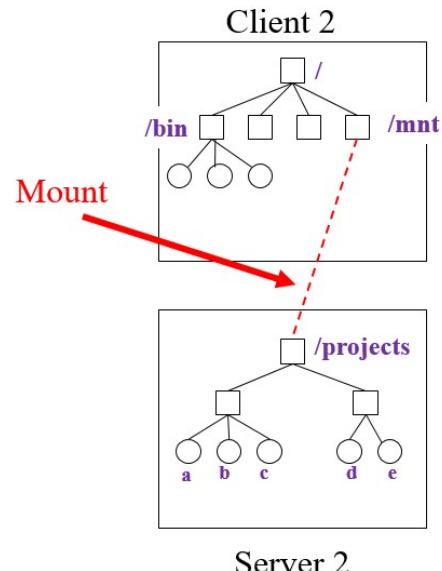
DFS - Distributed File Systems

- “**stateful**” remote file access
server keeps information about files opened by clients
 - o **more efficient** – no need to send to server all information about file each time clients use the file
 - o server may provide file locking system (synchronized usage of files) between clients
- “**stateless**” remote file access
 - o information about open files is kept on client machine
 - o **more immune to server crashes**



NFS protocols - mount

- Client asks to mount a directory *providing a host-name*
- Server returns a **file handle** to client, that contains:
 - File system type
 - Disk ID
 - i-node number / MFT record index / ... (depends on server OS)
 - Protection information
- Linux also support **automounting**
 - a set of remote directories (servers) is associated with a local directory
 - The first time a remote file is opened, Linux sends a message to each of the servers. The first one to reply wins, and its directory is mounted.
 - advantage – works even if some servers are down

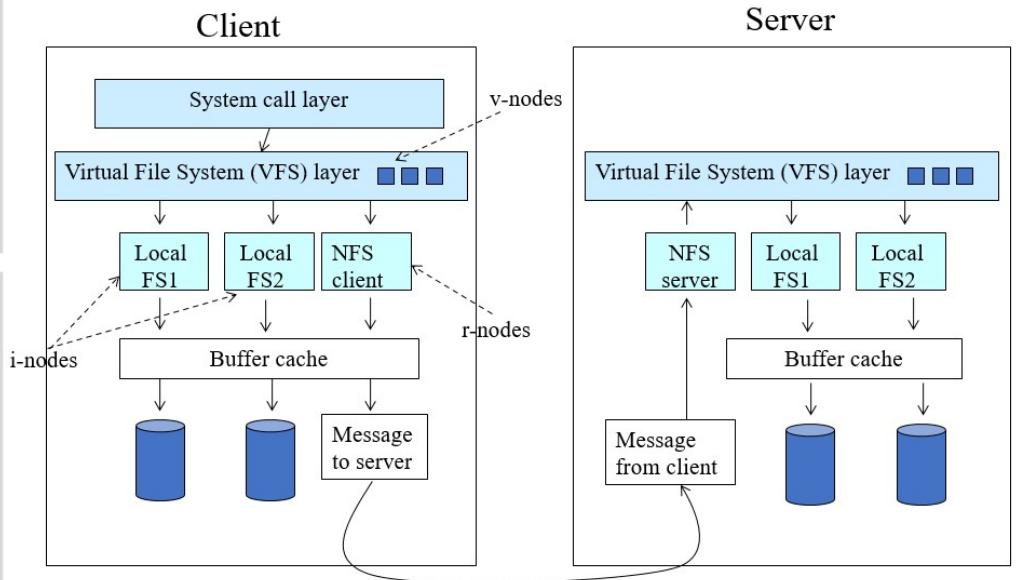


NFS protocols - file operations

- **open()** and **close()** files are executed only on client side
 - server *does not keep tables of open files*
 - crash of server *will not cause loss of information* (like file position, etc.) for clients
- **lookup()** is run on server, and returns a file handle to client
- **read()** and **write()** need absolute location of file block to read / write, since server does not know file's current position
- Unix file locking system cannot be used for mounted files

Layer structure of NFS

- Client and server have a Virtual File System layer (VFS), and an NFS module
- VFS keeps *v-nodes* for open files (virtual i-nodes)
- after mounting a remote file/directory, client creates *r-node* (remote i-node) in its internal table and stores the *file handle*
- A ***v-node*** points to either:
i-node (local file)
r-node (remote file)

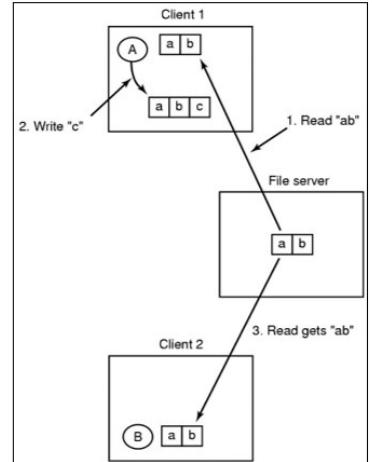
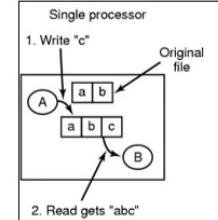


NFS: performance issues

- *Data sent in 8Kb chunks*
- **Read-ahead** – read ahead another 8Kb chunk even if not asked
- **Delayed-write** – write buffered until 8Kb chunk is full
 - when file closed – contents sent to NFS server immediately
- **Client-side buffer cache** is used
 - if two clients have same block in buffer, and one of them modifies the data, the second client would not see the changes immediately
– **inconsistency**

To solve this problem:

- cached block discarded: buffer containing data gets invalid after 3 seconds, buffer containing directory gets invalid after 30 seconds
- every 30 seconds, all dirty buffers are written back to server
- Whenever a cached file is opened, a message is sent to the server to find out when the file was last modified. If the last modification occurred after the local copy was cached, the cache copy is discarded, and the new copy fetched from the server.



ארQUITטורה של NFS

שתי NFS מיצאים אחת או יותר מתיקיותם לאישה מרוחק. כשתיקיה מסויימת הופכת לנגיעה, כל תת-התיקיות שלה גם הופכות לנגישות. רישימת התיקיות שהשרות מיצא מתחזקת בקובץ (לעתים /etc/exports), כך שניתן ליצא אותן באופן אוטומטי כאשר השירות מתחילה לעבוד.

לקחו NFS מעגנים את התיקיות שהשרות מיצא למערכת הקבצים שלהם באמצעות הכלי `mount`. התהיליך זהה מאפשר לקחו גישה ישירה לתיקיות והקבצים שהשרותים משתפים. לדוגמה,إمكان 1 יכול לחבר את התיקיה /bin של שרת 1 לתיקיה /bin שלו, כך שהוא יוכל להתייחס ל-`sh` shell כ/`bin` ולקבל את ה-`sh` shell של שרת 1. באופן דומה,إمكان 1 יכול לחבר את התיקיה/projects של שרת 2 לתיקיה /ast/work/usr שלו, כך שהוא יוכל לגשת לקובץ

a c /mnt/proj1/a. לקוח 2 יכול לחבר את התיקייה /projects ולגשת לקובץ a, אך ב/a1/a.mnt. מקום ההתקנה נקבע על ידי הלקוחות; השרת אינו יודע היכן התיקייה מותקנת בכל אחד מהלקוחות.

פרוטוקול NFS

הפרוטוקולים של NFS מגדירים את התקשרות בין שרתים ללקוחות. הפרוטוקול הראשון כולל עיגון (mounting).能夠 יכול לשלו לשרת שם נתיב ולבקש הרשאה לעגן את התיקייה בהיררכיה שלו. אם הנתיב הוא חוקי והתיקייה מיצאת, השרת מחזיר לקוח file handle שמצויה באופן ייחודי את סוג מערכת הקבצים, הדיסק, מספר node-ו של התיקייה,omidע אבטחה.

ב-*automounting*, ניתן להשתמש בכללי SMAutoMount שמאפשר התקינה אוטומטית של תיקיות מרוחקות. התקינה אוטומטית משפרת את האמינות והביטחונות במקורה שיש יותר שירות אחד, אך מחייבת שכל התקינות הרוחקות יהיו זהות.

הפרוטוקול השני של NFS הוא עבר גישה לתיקיות וקבצים. הלקוחות יכולים לשלו הודעות לשרתים כדי לנשל תיקיות ולקראן ולכתוב קבצים. הם גם יכולים לגשת למאפייני קובץ, כמו מצב קובץ, גודל, וזמן של השינוי האחרון. NFS אינם תומך בפקודות open ו-close. במקרה זאת, כדי לקרוא קובץ, לקוח שולח לשרת הودעת lookup עם שם הקובץ ובקשה לחפש אותו ולהחזיר ידית קובץ.

הדבר יכול להיעשות בשני דרכים: "stateless" או "stateful". בגישה הראשונה, השירותים שומרים מידע על הקבצים שנפתחו על ידי הלקוחות, מה שהופך זאת זה ליעיל יותר. שירותי עשיים גם לספק מערכת געילה קבצים לשימוש מתואם בין הלקוחות. בגישה לא מצבית, המידע על הקבצים הפתוחים מאוחסן במכונה של הלקוח, דבר שהופך אותו ליתר חסין לרירוסות שרת.

התכנית של NFS התקשתה במקור להשיג את הדיק של מערכת קבצים של *automount*. למשל, ב-NFS אין תמיינה לנעלית קבצים כי השירות לא יודע איזה קבצים פתוחים. NFS משתמש במנגנון ההגנה הרגיל של UNIX, עם הביטים אשו לבלים, לקובזה, ולאחרים. עם זאת כיום, ניתן להשתמש בקריפטוגרפיה מפתח ציבורי כדי לקבוע מפתח בטוח לאינומת הלקו והשרות בכל בקשה ותגובה.

IMPLEMENTATION OF NFS

רוב מערכות הלינוקס מיישמות את מערכת הקבצים הרשותית (NFS) באמצעות ארכיטקטורה של שלושה שכבות:

א. שכבת הקריאה למערכת (System-call layer): שכבה זו, הנמצאת בראש, מטפלת בקריאהות מערכת כמו close, open, read.

ב. שכבת מערכת הקבצים הווירטואלית (VFS): היא שומרת טבלה עם רשומה עבור כל קובץ שנפתחה. היא משתמשת ב"nodes-v" (איינודים וירטואליים) כדי לציין אם קובץ הוא מקומי או מרוחק, ומכללה מספיק מידע לגבייהם.

ג. מטמון הbufffer (Buffer cache): מספקת מטמון לנדרנים לשיפור ביצועים תוך שהוא משתמש שירות עם מערכת הקבצים.

כדי לפתח קובץ מרוחק, הליבה מזהה את הקובץ כמרוחק, ובתווך-node-v של הקובץ ומצאת את המצביע ל--node (איינוד מרוחק). לאחר מכן, לקוח NFS פותח את הקובץ, מקבל את ה-handle ויצר node-r בטבלאות

הפנימיות שלו לקובץ המרוחק. כל קובץ או ספרייה שנפתחים יש להם אnode-v המצביע ל-node-r או ל-node-i.

בצד הלוקו, מתחזקת מכתבת לקובץ המרוחק שמנפתחת ל-node-s על ידי טבלאות בשכבה ה-VFS. שום רשומות לא נשות הצד השרת. השרת פשוט מספק מכתבות לבקשת ובודק את התקיפות כאשר מתבצעת גישה.

ה-NFS יכולה לשפר ביצועים במספר מישורים. ראשית העברת נתונים בין הלוקו לשרת מתבצעת בחתיכות אדוות (בדרכם כלל 8192 בתים) למען היעילות. בנוסף, יש תcona בשם "read ahead" שמאפשרת שיפור ביצועים, שבה שכבת ה-VFS של הלוקו מבקשת את החתיכה הבאה של נתונים לאחר שהיא מקבלת את הנוכחות.

בנוסף המערכת משיגה ביצועים משופרים גם דרך caching. השרת מטמין נתונים כדי למנוע גישה לדיסק, בעודן מוחזקים מטמינים למאפייני קובץ (אינודים) ולנתוני הקובץ. אבל, caching יכול להביא לביעות עם עקביות, במיוחד ממספר לקוחות מטמינים אותו בлок קובץ. ביצוע NFS משתמש בטימרים הקשורים לבlok מטמון ובבדיקות רגילים עם השרת לגבי זמן השינוי האחרון של הקובץ כדי להפחית זאת.

לדוגמה בложים ממוטמנים מתנטלים לאחר תקופה מסוימת וכל החזצים המלוכלים מוחזרים לשרת כל 30 דקות. כמו כן, בכל פעם שקובץ ממוטמן נפתח, הלוקו שולח הודעה לשרת כדי לבדוק את זמן השינוי האחרון. אם הקובץ שונה לאחר המטמון, עותק המטמון נמחק וועתק חדש מתתקבל מהשרת.



מערכות הפעלה 2022

Operating Systems

Lecture 9 – File Systems

Dr. Marina Kogan-Sadetsky

נספח

UEFI

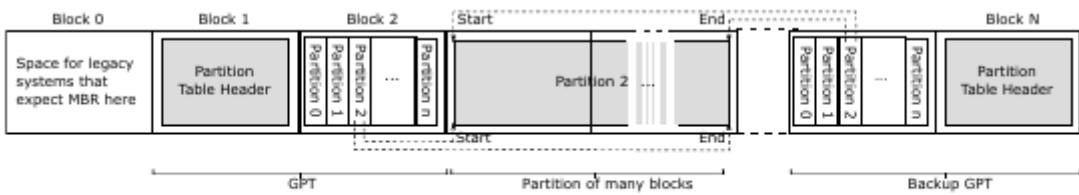


Figure 4-11. Layout for UEFI with partition table.

בxicom דנו בתחום ה-BOOT בפתיחת ה-MBR. האישה זו התגלתה כאיתית, אשר תלויות בארכיטקטורה ומוגבלת לדיסקים קטנים. מחשבים בני ימינו משתמשים בפתיחת ה-UEFI, או בשם המלא: **Unified Extensible Firmware Interface**. היא מאפשרת תחילת BOOT מהיר, תמייה בארכיטקטורות שונות ודיסקים בנפחים גדולים.

במקום שהמחשב יסתמך על ה-MBR בסקטור 0, הקשוחה (firmware) סורקת בטבלת המחיצות GPT אחר מחיצות מיוחדות שיש בהן מספיק מידע על מנת לבצע BOOT. הטבלה בנויה בצורה כזו שכל רשומה מכילה מידע על המיקום של המחיצה בדיסק ומזהה ייחודי, כולל הפניה לתחילת המחיצה והקצה שלה. בנוסף ישנו גיבוי של ה-GPT בבלוק האחרון באחסון.

المحיצות המיעילות ב-UEFI למשה מחליפות את ה-MBR. הן נקראות ESP (ראשי תיבות system partition) וונשענות על גבי מערכת הקבצים FAT. למעשה, תחילת ה-BOOT מוצגת על ידי תוכניות, קבצי האדרות וכו'. יתר על כן, הסטנדרט מניח כי הקשוחה יכולה להריץ תוכניות בפורמט מסוים, PE - דבר ההופך את הקשוחה למעין מערכת הפעלה קטנה.

העשרה - מימוש במערכות הפעלה

נתחיל בכך שבמהדרות מודרניות של וינדוס, אין תמייה ב-MBR כלל ולכן נתעלם מ-MBR. בעת התקנת מערכת הפעלה, בהנחה אין מחיצת ESP, נוצרת מחיצת ESP חדשה ובה נשמרים קבצים מיוחדים בעזרתם הקשוחה יכולה להעלות את וינדוס. הקבצים המיוחדים הללו נקראים Boot loader (מנהל התחול) וברגע שהמחשב נדלק ומוכן להתחלה תחילת ה-BOOT, הוא הולכת למעשה מריצ' את הקוד של ה-bootloader.

マイโครסופט בגיישתם תכננו את וינדוס כמערכת ההפעלה הבלעדית במחשב. הדבר בא לידי ביטוי במנהל האתחול של וינדוס, ה-`bootmgr`, שכן כל מה שהוא יודע לעשות זה להעלות וינדוס. בLINUKS הדבר נראה אחרת. ראשית, ישנו מספר תוכניות מנהלי אתחול הבולטות הן GRUB ו-`systemd-boot`, אך לצורך הדין מתמקד ב-GRUB. במחיצת ESP יש אפשרות לכל מנהל אתחול בנפרד (דבר המפשיט dualboot). אם לצורך העניין מותקן במחשב אובונטו, בתחילת ההתקנה, מתבצעת התקינה של grub, תחילת ההתקנה כולל העתקת הקבצים, התקנת מנהל האתחול במחיצת ESP והأدרטו. במהלך תחילת ההגדרה grub מאתרת אחר קיומם של מערכות הפעלה אחרות במחיצת ESP.

בעת עליית המחשב בו הותקן ubuntu, עולה תפריט בחירת תוכניות אשר ניתנות להרצה על ידי ה-firmware. אם לצורך העניין המשמש מעוניין בוינדוס, grub הולכת למעשה מרוץ תריצ' את מנהל האתחול של וינדוס אשר ישלים בלבד את תחילת ה-`boot` של וינדוס.

סטרטגיות גיבוי מערכות קבצים נוספת

דחיסת מידע: כדי לפחות בגודל האחסון, המידע יכול לעבור דחיסה לפני הגיבוי עצמו. עם זאת, יש להימנע מאלגוריתמי דחיסה שעולים לאורם למידע המגובה לא להיות בר קרייה, אפילו אם יש שגיאה אחת.

צילום מצב גיבוי מערכת קבצים פעילה יכול להיות מסובך לגיבוי ללא חוסר עקביות. צילום מצב כולל יצירת תמונה של מערכת הקבצים, דבר המאפשר גיבויים עקביים בהמשך.

אבטחה אהגיבויים צריכים להישמר בצורה בטוחה, גם פיזית וגם דיגיטלי. שמירת הגיבויים במקום מרוחק הוא קריטי כדי להגן מנזקים כמו שרפות וגישה לא מורשת לגיבוי.

טכנית נוספת להפחיתה התנוועה של הדיסק קשה

קיבוץ בЛОקים

כאשר נדרש כתיבה לקובץ, מערכת הקבצים בדרך כלל מנסה בлок אחד כל פעם. כאשר מערכת הקבצים עוקבת אחר הבלוקים החופשיים בעזרת bitmap והוא נמצא בזיכרון הראשי, קל להקנות בлок חופשי הקרוב לבלוק הקודם - דבר המפחית את המרחק שהזרוע צריכה לעבור.

אם חלק מרישימת הבלוקים הפנויים נמצא בדיסק, קשה יותר לאתר בלוקים סמוכים, ולכן מערכות כאלו ינהלו את הדיסק בקבוצות של בלוקים סמוכים במקום בלוקיםבודדים. אם כל סקטור מכיל 512 בתים, המערכת יכולה להקנות אחסון ביחידות של 2 בלוקים. כאשר הדיסק מעביר את שאר ה-1KB, קריית הקובץ בצורה סדרתית כאשר המערכת לא מבצעת שום דבר נחפה ליעילה יותר. השיפור בזמן האיתור הוא בחלוקת של 2.

העשרה

בחלק זה ברצוני להביא שיקולים נוספים של מערכות קבצים ומימושים נוספים אשר לא נלמדו בהרצאות.

Journaling File Systems

מערכות קבצים אלו מוצבו בצורה מיוחדת על מנת לשפר את אמינות מערכות הקבצים הרגילות ואת יכולת השחזור שלהן. מערכות קבצים אלו מחזיקות ביוםן המתעד פעולות קרובות עוד לפני שהן בוצעו. ברגע שהמערכת קורסת, מערכת הקבצים יכולה להתיחס ליום זה במהלך השחזור על מנת לסייע תהליכי מתוכננים או לשחרר תהליכי קודמים.

נראה בתרור דוגמה פשוטה כמו מחיקה של קובץ. במערכת קבצים המחזיקה יומן הפעולות הבאות יכולות להתרחש:

- א. יצירת רשומה חדשה ביוםן המתארת את שלבי מחיקת הקובץ.
- ב. כתיבת הרשומה בדיסק תוך כדי ידוא שלאה.
- ג. ביצוע פעולה מחיקת הקובץ.
- ד. בעת סיום תקין של הפעולה, רשומת היום נמחקת.

אם המערכת קרסה לפני שפועלת המחיקה הסתימה, מערכת הקבצים יכולה להתבונן ביוםן כדי לוודא שהפעולה הסתיימה בהצלחה.

מערכות קבצים נפוצות מהסגן הזה הם NTFS ו-ext4.

Linux File System

لينوكס תומכת במערכות קבצים רבות, כולל מערכות קבצים מקומיות ומערכות קבצים מרוחקות. לשם כך לינוקס מגדרה משnek נוח לתהליכי המסתיר את פרטיה המימוש של מערכת הקבצים. המשך זה הוא שכבת מערכת הקבצים הווירטואלית, כפי שדרנו בה בחלק על NFS. אנו נדונם במערכות הקבצים EXT2 EXT4 ו-EXT4.

EXT2 (Extended File System 2):

מערכת הקבצים EXT2, שחקה תפקיד מרכזי באבולוציה של מערכות הקבצים בלינוקס. מערכת קבצים זו מחלקת את הדיסק לקבוצות של בלוקים, כאשר כל קבוצה בлокים כולן מחזיקה ברכיבים מסוימים. קבוצות אלו כוללות את ה-block super אשר מכיל מידע על מערכת הקבצים, רשומות המתארות מיקומים של bitmaps וסטטיסטיקות, bitmaps למעקב אחר בלוקים פנויים -aps על מעקב אחר nodes פנויים.

בדומה לינוקס, לב ליבה של מערכת הקבצים הוא ה-node, מבנה באורך 128 בתים המתאר קובץ, כולל גודל הקובץ ומיקומי הבלוקים. על מנת להפחית פרוגרנטי糍ה פנימית, מערכת הקבצים מנסה לקבץ קבצים באוטה קבוצה של בלוקים כמו תקיית האב. כמו כן היא משתמשת ב-bitmaps על מנת לקבוע היכן להקצות מידע חדש ואף להקצות מראש בלוקים כדי לצמצם פרוגרנטי糍ות עתידית.

כמו כן היא מחזיקה במתמון תיקיות על מנת ליעיל את תהליך החיפוש אחר קבצים בתיקיות גדולות.

EXT4

מערכת הקבצים 4 EXT היא מערכת הקבצים הכי נפוצה בימינו, והיא ברירת המחדל ברוב הפצצות הלינוקס, כולל דיביאן. השכלול המרכזי במערכת קבצים זו היא התמודדות עם אובדן מידע לאחר קריסת מערכת או נפילות חשמל בעזרת ניהול יומן.

מערכת הקבצים מתחזקת יומן של פעולות. השינויים רשומים בצורה סידרתית ולאחר מכן נשמרות בדיסק. במקרה של קריסה, המערכת יכולה להتاושש על ידי ביצוע מחדש של רשומות היומן. היומן עובד בצורה של באפר מעגלי, כאשר פעולות קריאה וכתיבה נשמרות בעזרת רכיב אחר, ה-JBD.

ה-JBD, ר"ת Journaling Block Device מחזיק ב-3 מבני נתונים: רשימת יומן, רשומות level low, ה-handler מקבץ פעולות אוטומטיות רלוונטיות והטרנסקציות שומרות רשומות יומן בצורה רציפה.

בנוסף, EXT4 הציגה את השימוש ב-extents, המתאר בלוקים רציפים באחסון. בכר 4 EXT מפחיתה בפעולות לכל בלוק באחסון, דבר המפחית את ה프로그רנטי糍ה עבור קבצים גדולים ומאפשר לתמוך בקבצים גדולים יותר.