

מערכות הפעלה מטרת הקורס

□ הבנה של עקרונות התכנון והשימוש במערכות הפעלה.

□ יסודות של ניתוח ביצועים של מערכות הפעלה.

□ יסודות של תכנות מערכת הפעלה.

□ עקרונות של תכנון מערכות תוכנה גדולות.

מהי מערכת הפעלה?

□ שכבת תוכנה לניהול והסתרת פרטים של חומרת המחשב.

□ מספקת לאפליקציה אבסטרקציה של מכונה וירטואלית ייעודית וחזקה (זיכרון עצום, מעבד ייעודי חזק מאוד...)
"Computer Science is the art of abstraction"

□ מנהלת את משאבי המערכת ומשתפת אותם בין תהליכים, תוכניות, ומשתמשים.

□ ממשק נוח למשתמש SHELL

תפקיד אצרכת ההפעלה

□ מאפשרת להריץ אפליקציות

□ מבטיחה נכונות.

■ גבולות זיכרון

■ עדיפויות

■ מצב יציב

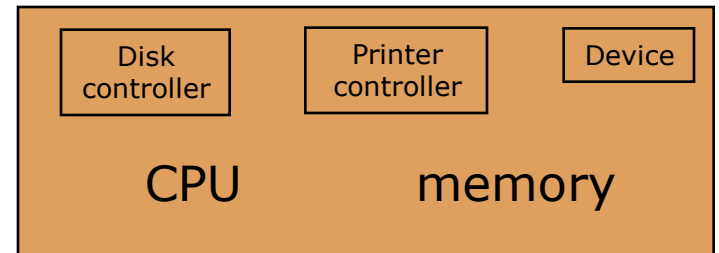
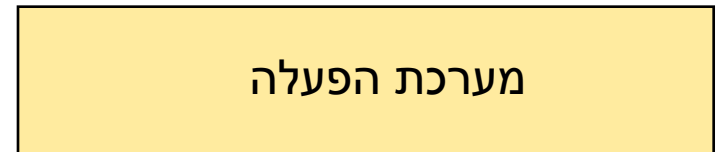
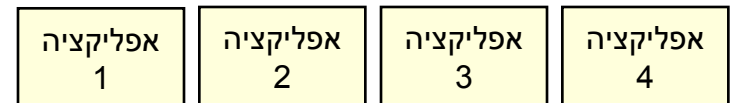
□ מספקת נוחיות.

■ הסתרת פרטים

■ תיאום

■ קריאות מערכת-הפעלה

■ מערכת קבצים



עיתון משאבים

אפליקציה רוצה את כל המשאבים:

■ זמן מעבד

■ זיכרון

■ קבצים

■ אמצעי קלט / פלט

■ שעון

מערכת ההפעלה נותנת לכל אפליקציה **אשליה** של מערכת שלמה משל עצמו.

מטרות מערכת ההפעלה

- מאפשרת למשתמשים לבצע תכניות באופן:
 - איכותי: לספק את השירותים הנדרשים באופן מהיר ויעיל
 - יעיל: למקסם את הניצול של משאבי המערכת.
 - למקסם את מספר המשתמשים המקבילים שירות מהמערכת.
 - מתן זמני תגובה מקובלים במערכת.
 - הוגן: מחלקת את משאבי המערכת באופן הוגן.
 - נוח: חוסכת למשתמש את הצורך לדעת את הפרטים השונים של החומרה והאמצעים הנדרשים בעת קבלת שירותי מערכת.
 - נכון: תוכניות לא משפיעות על תוכניות אחרות, שהטיפול וקבלת השירות ממספר רב של התקני קלט/פלט יהיה נכון, שהמערכת תשמר במצב תקין וכו'.
- להגן על משאבי המערכת ולתת שירות רק למשתמשי המערכת.
- מערכת ההפעלה כמכונה מדומה או כמכונה מורחבת.

סולאי שירות שונים נדרשים

במערכת

- מערכת לניהול תהליכים.
- מערכות לניהול קבצים.
- מערכות לניהול גישות לזיכרונות המשניים.
- מערכות לניהול משאבים חיצוניים.

תתי - מצרכות

□ ניהול תהליכים

- יצירה/השמדה של תהליכים
- השהייה / חידוש פעילות של תהליך
- מנגנונים לתקשורת בין תהליכים
- טיפול במצבי שגיאה של תהליכים

□ ניהול מעבד ראשי

- הקצאה ושחרור של מעבד
- הרציפות המדומה של תהליך
- סנכרון תהליכים

תתי אצרכות (המשק)

□ ניהול זיכרון ראשי / מטמון

■ הקצאה/שחרור של זיכרון

■ עקיבה אחר חלקים פנויים בזיכרון

■ עקיבה אחר חלקי זיכרון תפוסים: כתובות, צרכן, סוג מידע

□ ניהול קבצים

■ יצירה/השמדה של קבצים מסוגים שונים

■ ניהול מערכת עקיבה לקבצים

□ ניהול זיכרון משני

■ ניהול שטחי אחסון פנויים/פגומים/תפוסים

■ הקצאה/שחרור שטחי אחסון

תתי אצרכות (המשק)

□ מערכת קלט/פלט

- הקצאה, עקיבה, שחרור של מאגרי זיכרון, תורים וקבצי ביניים

- ניהול של drivers ספציפיים לחומרה נתונה

□ הגנת מערכת

- הרשאת גישה ואובייקטי המערכת ולשרות רק לצרכנים מורשים

- טיפול במצבי שגיאה וכשל

- טיפול במצבים של אובדן משאבים

- טיפול במצבים של עומס על משאבי המערכת

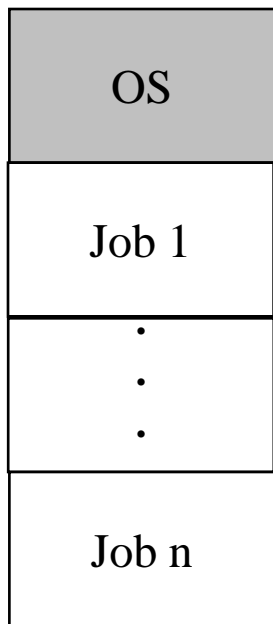
תתי מצרכת (המשק)

□ תקשורת

- טיפול בערוצי תקשורת שונות
 - ניהול הבקשות לגישה למדיה השונה
 - כולל טיפול בניהול מערכות קלט/פלט
- ## □ ניהול סביבות עבודה לצרכנים השונים
- כלים ואמצעים
 - קונפיגורציית משתמש
 - ממשק (גרפי) אדם-מכונה

מערכות ריבוי תוכניות ושימוש זמנים

- מספר תוכניות לביצוע היושבות בדיסק באותו הזמן
- קבוצת "תהליכים" המועתקת מהדיסק לזיכרון הראשי



- בחירה של תהליך לביצוע
- ניהול התהליכים לביצוע
- מושג Context Switching
- ניצולת משאבים משופרת
- התפתחות מערכת שיתוף זמנים

חלוקת הזיכרון הראשי

רציפות המדומה של התהליך

□ במערכת מרובת תהליכים יכול תהליך לבקש מספר פעמים פעולות קלט/פלט - < לרצות שירות עבוד מספר פעמים

□ תפקיד מערכת ההפעלה הוא לחסוך מהתהליך אינפורמציה לא רלוונטית לריצתו והנוגעת לאופן ניהול המשאבים במערכת.

■ כלומר יש לתת לתהליך את "האשלייה" כאילו רק הוא נמצא במערכת

□ המערכת עושה זאת באמצעות "צילום" מצב התהליך מיד לפני צאתו את המעבד המרכזי ושחזור מצב זה עם כניסתו מחדש למעבד.

התפתחות מערכות הפעלה

- חומרה יקרה ואיטית, כוח-אדם זול
 - Batch jobs, ניצול החומרה 24x7: IBM S/360
- חומרה יקרה ומהירה, כוח-אדם זול
 - Unix: Interactive time-sharing
- חומרה זולה ואיטית, כוח-אדם יקר
 - מחשב אישי לכל משתמש: MS-DOS

ההתפתחות של מערכות

הפעלה (1)

- **מערכת Desktop למחשב אישי (personal computer system)** מיועדות למשתמש יחיד בזמן נתון, המריץ הרבה יישומים, online, interactive
- **מערכת אצווה** מריצה את התוכניות (עבודות, Jobs) בזו אחר זו. העבודה שרצה שולטת במשאבים וכל האחרות ממתינות.
- **מערכת אצווה מרובת תוכניות (multi-programming Batched system):** בזמן שתוכנית אחת משתמשת באמצעי קלט/פלט – אחרת מקבלת את המעבד.

ההתפתחות של מערכות הפעלה (2)

□ **מערכות לשיתוף זמנים (Time sharing):** זמן המעבד מוקצה למשימה (job) הנמצאת בזיכרון באותה עת. המשימות מתחלפות ביניהן. דגש על אינטראקטיביות.

► **מערכות מקביליות:** מחשב מקבילי הוא מחשב המצויד ביותר ממעבד אחד, עם תקשורת צמודה. המעבדים חולקים זיכרון ושעון. התקשורת נעשית לרוב דרך הזיכרון המשותף.

ההתפתחות של מערכות

הפעלה (3)

□ **מערכות מבוזרות:** במערכות אלו החישוב מבוזר בין מספר מעבדים פיזיים. לכל מעבד יש את הזיכרון המקומי שלו. המעבדים מתקשרים ביניהם דרך קווי תקשורת מסוגים שונים, כמו אפיקים (buses) מהירים או קווי טלפון.

□ **מערכות זמן אמת RTOS :** מערכות מחשב אשר בהן יש דרישות לביצועים בזמנים מסוימים, ולכן יש שימוש בהוראות המוגבלות בזמן ביצוע.

הוֹוֶה וְאַתִּיד

□ חומרה זולה מאוד, כוח חישוב רב.

- ריבוי משימות: Windows NT, Windows Vista, Linux, Solaris, BSD, Mac OS X
- ריבוי מעבדים וריבוי ליבות (multi-core)
- שיתוף משאבים בסיסי: דיסקים, מדפסות, ...

□ רשתות מהירות.

- הרשת היא המחשב: SETI@home, Grid Computing
- הרשת היא אמצעי אחסון: SAN, Web storage

הצמיח הקרנף

□ וירטואליזציה - סגירת מעגל?

- ניתוק מערכת ההפעלה מהחומרה
- מספר "מחשבים מדומים" על-גבי מכונה פיזית אחת
- בשילוב רשתות מהירות: Cloud Computing
- מערכת הפעלה ייעודית - Software Appliance

□ מזעור והטמעה

- טלפון סלולרי כמערכת מחשב, Netbooks
- מחשוב בכל מקום - Pervasive/Ubiquitous Computing

חלקי מערכת ההפעלה

□ החלקים העיקריים של מערכת הפעלה הם:

■ ליבה (גרעין, kernel) - שכבת התוכנה אשר אחראית על

הקשר שבין שכבת התוכניות אל שכבת החומרה

■ ממשק תכנות יישומים (Application Programming -

API) נותנת למשתמש הקצה את האפשרות להריץ פקודות של

מערכת ההפעלה

מנה האשה

התנהגות מערכת ההפעלה מוכתבת (חלקית) על-ידי
החומרה שעליה היא רצה

■ סט פקודות, רכיבים מיוחדים

החומרה יכולה לפשט / לסבך משימות מערכת ההפעלה

■ מחשבים ישנים לא סיפקו תמיכה לזיכרון וירטואלי

■ מחשבים מודרניים מכילים ריבוי ליבות ותמיכת חומרה בריבוי
תהליכים

מנאנוני חומרה לתמיכה במערכת ההפעלה

- שעון חומרה
- פעולות סנכרון אטומיות
- פסיקות
- קריאות מערכת-הפעלה
- פעולות בקרת קלט / פלט
- הגנת זיכרון
- אופן עבודה מוגן (protected)
- פקודות מוגנות

פקודות מולאנות

□ חלק מפקודות המכונה מותרות רק למערכת-ההפעלה

- גישה לרכיבי קלט / פלט (דיסקים, כרטיסי תקשורת).
- שינוי של מבני הנתונים לגישה לזיכרון (טבלת דפים, TLB).
- עדכון של סיביות **מוד** (מצב) מיוחדות (לקביעת עדיפות טיפול בפסיקות).
- פקודת halt.

עבודה במצב מואן

□ הארכיטקטורה תומכת בשני מצבים לפחות:

■ kernel mode

■ user mode

(במעבדי IA32 יש ארבעה מצבים...)

□ המצב נשמר באמצעות status bit ברגיסטר מוגן.

■ תכניות משתמש רצות ב-user mode.

■ מערכת ההפעלה רצה ב-kernel mode.

□ המעבד מבצע פקודות מוגנות רק ב-kernel mode.

אז איך מעמם ניאם פדיסק?

קריאה לפרוצדורת מערכת-הפעלה (*system call*)

□ קריאה לפסיקה:

■ מציב ברגיסטר `eax` את מספר ה- `system call` המבוקש.

■ מכין פרמטרים לפי השרות המבוקש

■ מבצע פקודת "`int 0x80`"

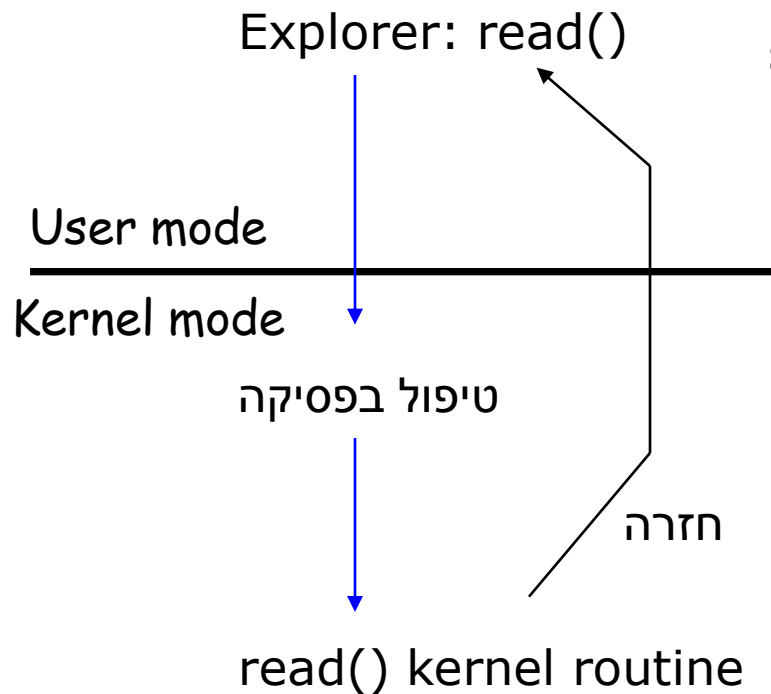
□ פרמטר מזהה את קריאת המערכת

□ שומרת את מצב התוכנית הקוראת

□ מוודאת את הפרמטרים (למשל, מצביעי זבל)

□ רוטינת השירות במערכת ההפעלה

□ חזרה לתוכנית הקוראת כאשר מסיימים



דואל: Linux עם מעבדי Intel

□ אתחול:

- טבלת מזהי פסיקה (Interrupt Descriptor Table) עם מטפלים (handlers) לכל אחד מסוגי הפסיקות.
- ווקטור 128 (= 0x80) מתאים ל-system calls.
- לקוד גרעין יש עדיפות 0, לקוד משתמש עדיפות 3.

□ כניסה ב IDT, המתאימה לווקטור 128, מכילה:

- מצביע לקטע קוד גרעין המטפל ב-system calls
- אישורים עבור קוד עם עדיפות 3 ליזום קריאה לפסיקה זו.

□ בביצוע system call, תהליך המשתמש:

- מציב ברגיסטר eax את מספר ה-system call המבוקש.
- מכין פרמטרים נוספים לפי השרות המבוקש
- מבצע פקודת "int 0x80" (פסיקה יזומה ע"י תוכנה).

האנה אף הליכרון

□ מערכת ההפעלה צריכה להגן על תוכניות המשתמשים, זו מפני זו (עם או בלי כוונה רעה).

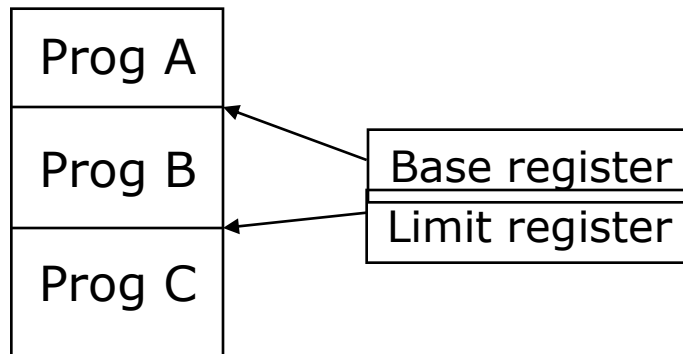
□ מערכת ההפעלה צריכה להגן על עצמה מפני תוכניות המשתמשים.

■ ועל תוכניות המשתמשים מפניה?

□ שיטה פשוטה: base register, limit register לכל אחת מהתוכניות.

■ מוגנים בעצמם.

□ זיכרון וירטואלי.



יום בחיית של מערכת ההפעלה

עלייתה של מ.ה. □

□ בהדלקת המחשב מורצת תכנית הנקראת טוען העלייה (bootstrap program/loader).

□ תכנית זאת מזהה את החומרה הנכללת במערכת, רכיבי הציוד המחשב (המעבד ואוגריו, הזיכרון), בודקת את תקינותה, מאתחלת אותה. ב PC מכונה תכנה זאת בשם BIOS

□ BasicI/O System. בסיום האיתחולים, יטען טוען העלייה את גוש העלייה של מ.ה. (boot block), והוא יטען את יתר גרעין המערכת. בדרך כלל זה Master Boot - MBR
□ 0-Record סקטור של הדיסק.

□ מלבד אתחול המערכת, נכנסים לגרעין רק בגלל **מאורע**.

□ הגרעין מגדיר אופן טיפול בכל מאורע.

■ חלק נקבע על ידי ארכיטקטורת המעבד.

■ מנגנון כפי שראינו.

□ פסיקות ו-exceptions:

■ Interrupts (פסיקות) נגרמות על-ידי רכיבי חומרה (שעונים, סיום ק/פ)

■ Exceptions מגיעות מהתוכנה (פקודה מפורשת, page fault)

רכיבי אצרכת ההפעלה

- תהליכים
- זיכרון
- קלט / פלט
- זיכרון משני
- מערכות קבצים
- הגנה
- ניהול חשבונות משתמשים
- ממשק משתמש (shell)
- זאת ועוד...
- אתחול
- גיבוי
- ...
- דפדפן?

ארכאון מערכת ההפעלה

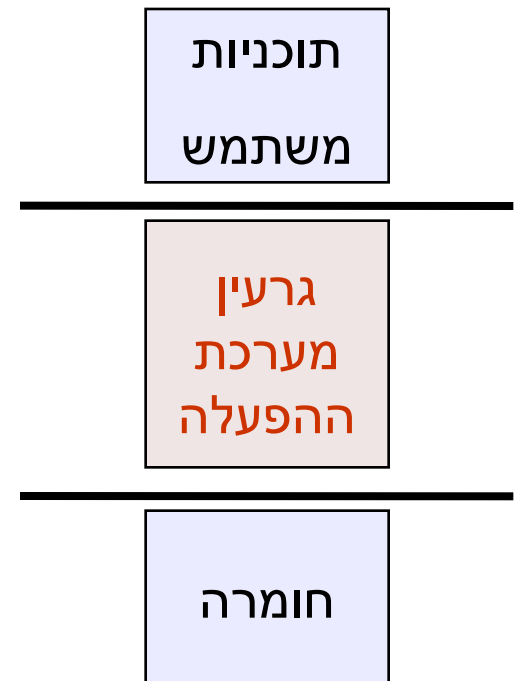
✓ תקשורת זולה בין מודולים

✗ קשה להבין

✗ קשה לשנות או להוסיף רכיבים

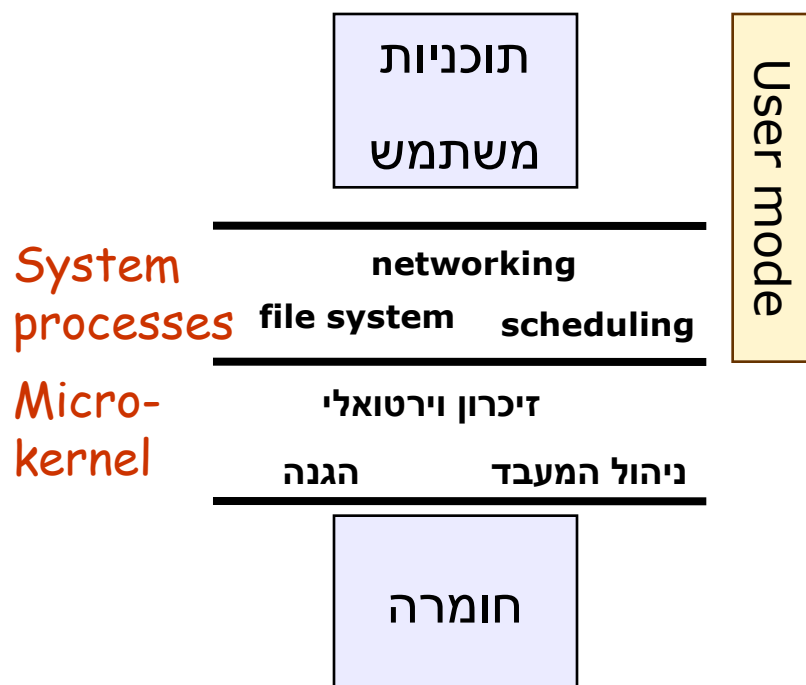
מה האלטרנטיבה?

בראשית... מונוליתית



ארכיון מערכת ההפעלה

אח"כ... גרעין קטן

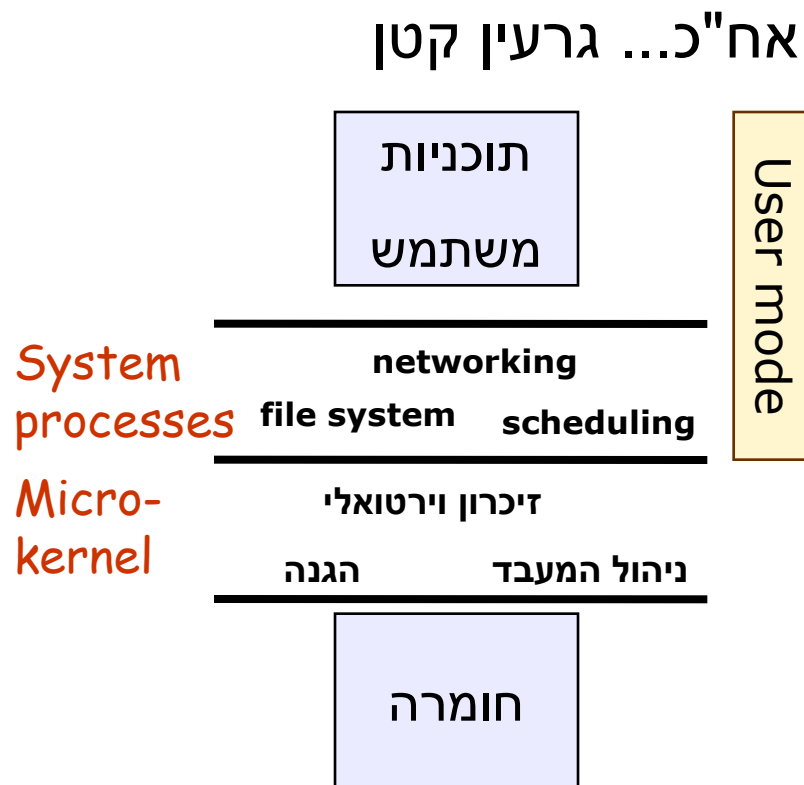


בראשית... מונוליתית



מערכת מונוליתית היא למעשה אוסף גדול של פונקציות שונות. מערכות מרבדות **layered systems** מערכות שמרכיבין מאורגנים ברבדים אשר נמצאים ביחס היררכי. לכל רובד מוגדר ממשק שדרכו הרבדים הסמוכים בהיררכיה יכולים לתקשר ביניהם.

ארכאון מורכבת ההפעלה



Micro Kernel: שכבה דקה
מספקת שירותי גרעין

אמינות גבוהה יותר ✓

קל להרחיב ולשנות ✓

ביצועים גרועים (מעבר בין
user-mode לבין kernel-
mode)

דוגמאות:

Mach, OS X, ~Windows NT

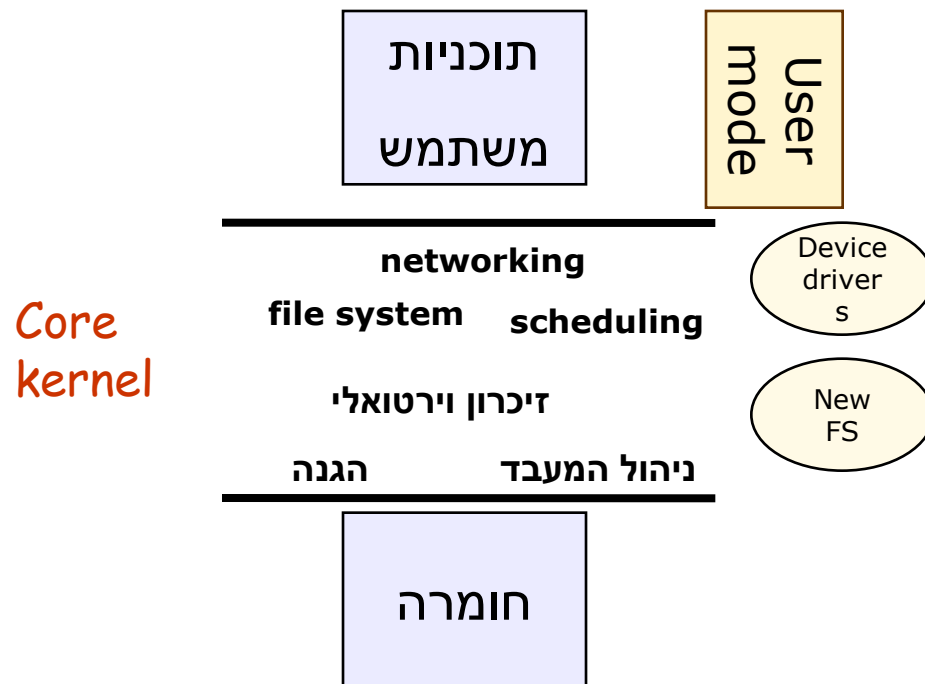
virtual machine מכונה מדומה - אחד התפקידים של מערכת הפעלה הוא אספקה של שירותים של המכונה המורחבת ושל המכונה המדומה. ניתן להפריד את שתי הקונספציות הללו ולראות את מערכת ההפעלה כשכבת תוכנה שמספקת לשכבות שמעל מספר העתקים של החומרה עם כמות משאבים פחותה מזו שבחומרה הפיזית בפועל. במובן מסוים זו אינה מערכת הפעלה רגילה, שכן על כל מכונה מדומה יכולה לרוץ מערכת הפעלה חדשה, ואולי גם אותה מערכת שתשכפל את עצמה.

מערכות Exokernel - ניהול זיכרון, תזמון תהליכים ותקשורת הוצאו אל מחוץ לגרעין. ההחלטה הזאת נובעת מכך שמערכת ההפעלה איננה כופה על המשתמש את צורת השימוש במשאבים. מערכת ההפעלה דואגת רק להגנה על המשאבים ולניהולם. המערכת מקצה משאבים לתכניות משתמש שיכולות לעשות שימוש במשאבים אלו בצורה ייחודית רק להן. כל תכנית יכולה להשתמש בספריות מוכנות המממשות אבסטרקציות כגון קבצים וזיכרון מדומה או, לחלופין, exokernel יכולה להשתמש במשאבים בצורה ייחודית באמצעות ספריות ייעודיות.

מערכות שרת-לקוח client-server מבוססות על גרעין מערכת ההפעלה מתפקד כדוור-גרעין מינימליסטי- **micro-kernel** המריץ תהליכים אל שרתים עצמאיים (תהליכים גם הם), המפוזרים במערכת, כגון שרתי מערכת הקבצים, מנהל הזיכרון, תכנית התקשורת, ותכנית המסך. כאשר מתקבלות תשובות מהשרתים, הגרעין מחזיר את התשובה לתהליך המבקש. בשיטה זו, התוכנה מורכבת מיחידות עצמאיות, שכל אחת מהן ממלאת תפקיד מוגדר. עם כל יחידה כזו ניתן להידבר רק באמצעות מספר מוגבל של הודעות, דרך ערוץ תקשורת פנימי או חיצוני כלשהו. שיטה זו מצמצמת מאוד את התלות בחומרה. ההבדל הוא ש Exokernel מקצה חלק ממשאבי המערכת לטובת המשתמשים אשר משתיתים על המשאבים הגולמיים האלה אבסטרקציות שלהם, ואילו במערכות שרת-לקוח כל סוג משאבים מנוהל על ידי תוכנת לקוח אחת שמספקת אבסטרקציה אחידה של המשאב לכל המשתמשים.

ארכאון אצרכת ההפעלה

היום... גרעין מודולרי



מודולים דינמיים, שניתן לטעון

ולהסיר אותם מהזיכרון

✓ מאפשר לטעון רכיבי קוד

ונתונים לפי דרישה

✓ מונע מהגרעין "להתנפח"

ללא צורך בזיכרון המחשב

✓ קל להרחיב ולשנות

✓ ביצועים טובים

דוגמאות:

Solaris, Mac OS X,

Linux (kernel modules),

Windows (Dynamic device Drivers)

תהליכים

- מהו תהליך?
- מבני הנתונים לניהול תהליכים.
- החלפת הקשר.
- ניהול תהליכים ע"י מערכת ההפעלה.

תהליך

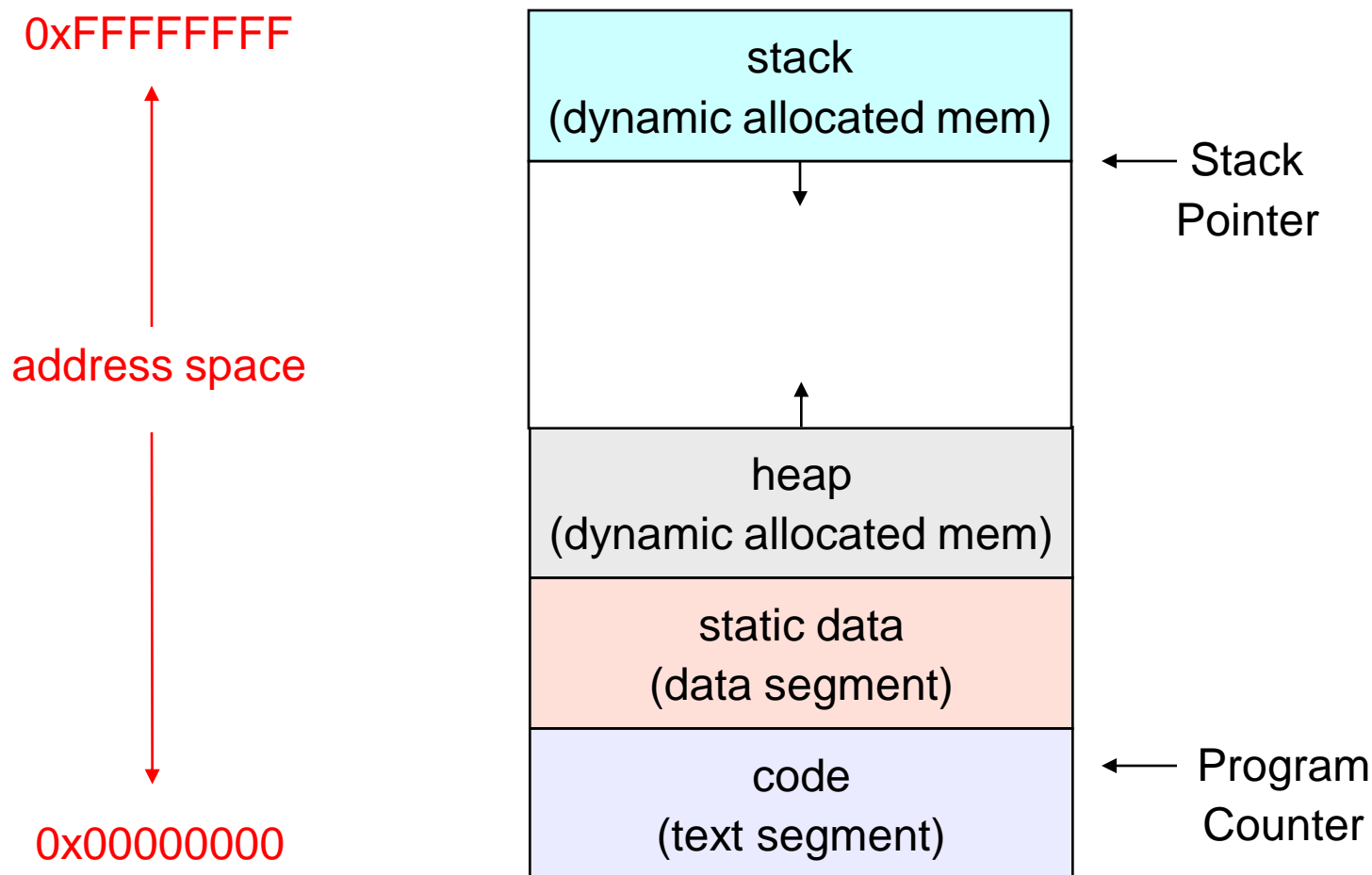
□ אבסטרקציה:

- יחידת הביצוע לארגון פעילות המחשב
 - יחידת הזימון לביצוע במעבד ע"י מערכת ההפעלה
 - תוכנית בביצוע (סדרתי = פקודה-אחת-אחר-השנייה)
- נקרא גם job, task, sequential process

מה מאפיין תהליך?

- מרחב כתובות
- קוד התוכנית
- נתונים
- מחסנית זמן-ביצוע
- program counter
- רגיסטרים
- מספר תהליך (process id)
- תהליך לעומת תוכנית:
- תוכנית היא חלק ממצב התהליך
- תוכנית יכולה לייצר כמה תהליכים

מרחב הכתובות של התהליך



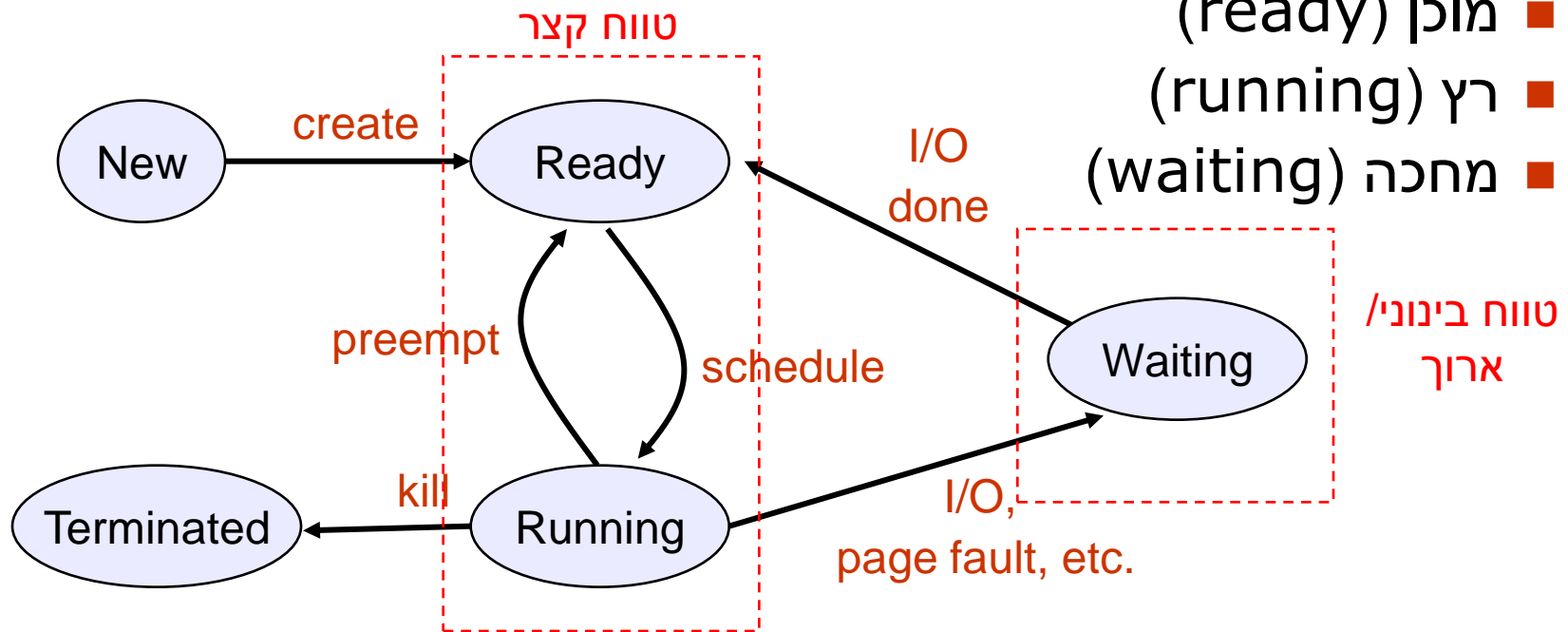
מצבי התהליך

□ כל תהליך נמצא באחד המצבים הבאים:

■ מוכן (ready)

■ רץ (running)

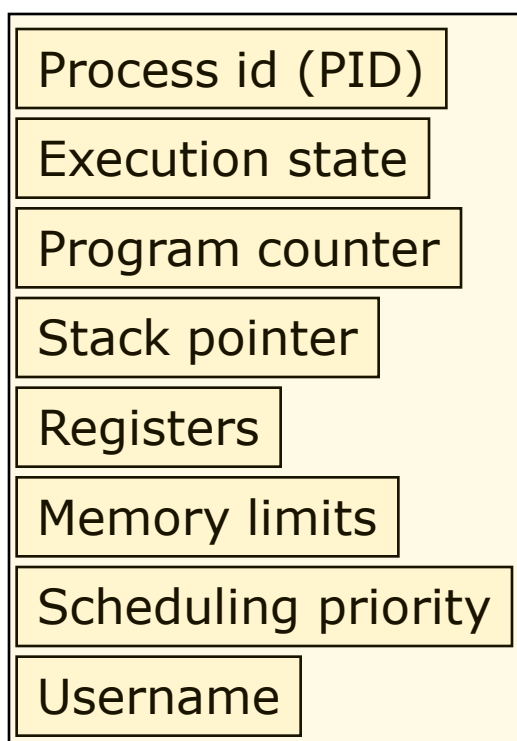
■ מחכה (waiting)



□ מתי מזמנים / מפנים תהליכים?

מבני הנתונים של תהליך

ב Linux, מכיל 95+ שדות!



Open files
status

□ בכל זמן, הרבה תהליכים פעילים במערכת

□ לכל תהליך **מצב**

□ Process control block (PCB) שומר את מצב התהליך כאשר אינו רץ.

■ נקרא Process Descriptor ב-Linux

□ נשמר כאשר התהליך מפונה, נטען כאשר התהליך מתחיל לרוץ.

מצב המצב וה-PCB

□ כאשר תהליך רץ, המצב שלו נמצא במעבד:

■ PC, SP, רגיסטרים

■ רגיסטרים לקביעת גבולות זיכרון

□ כאשר המעבד מפסיק להריץ תהליך (מעבירו למצב

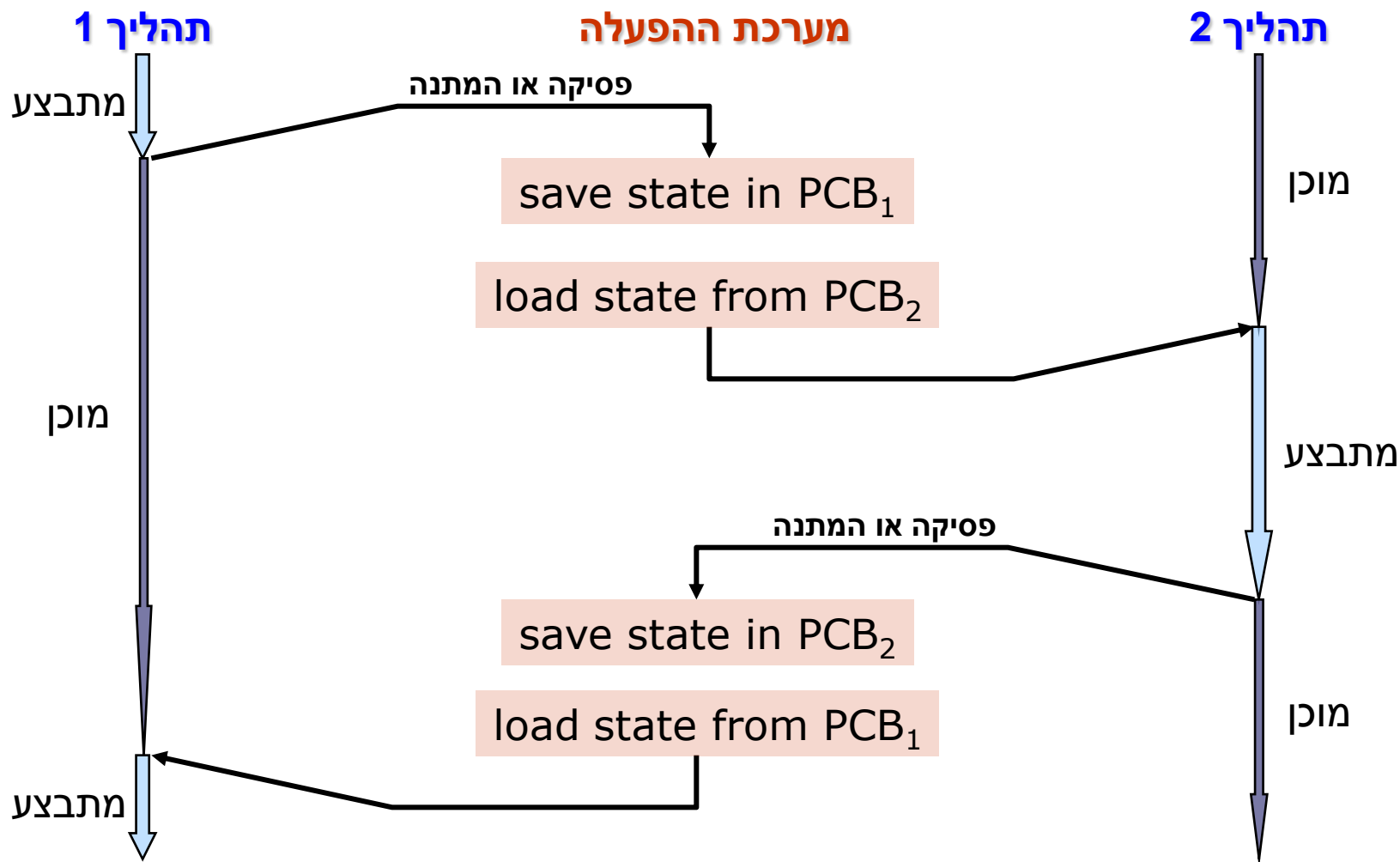
המתנה), ערכי הרגיסטרים נשמרים ב-PCB.

□ כאשר המעבד מחזיר תהליך למצב ריצה, ערכי

הרגיסטרים נטענים מה-PCB.

Context Switch: העברת המעבד מתהליך אחד לשני.

Context switch

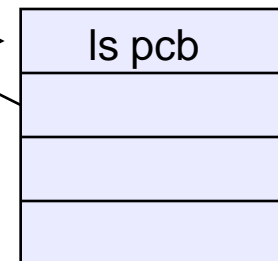
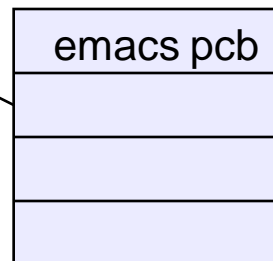
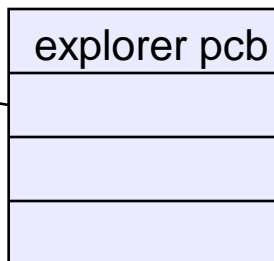
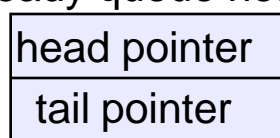


תורי מצבים

□ מערכת ההפעלה מחזיקה תורים של תהליכים

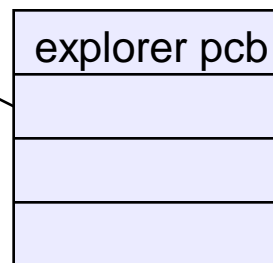
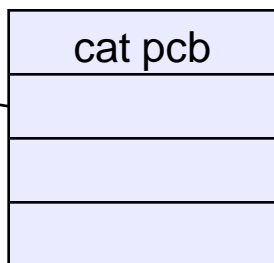
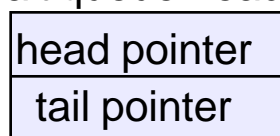
■ תור (או מבנה נתונים מתוחכם יותר) ready

Ready queue header



■ תור waiting, למשל ל-device מסוים

Wait queue header



ה-PCB ותוכי האצבים

ה-PCB הוא מבנה נתונים בזיכרון מערכת ההפעלה.

- כאשר התהליך נוצר, מוקצה עבורו PCB (עם ערכי התחלה), ומשורשר לתור המתאים (בדרך-כלל, ready).
- ה-PCB נמצא בתור המתאים למצבו של התהליך.
- כאשר מצב התהליך משתנה, ה-PCB שלו מועבר מתור לתור.
- כאשר התהליך מסתיים, ה-PCB שלו משוחרר.

יצירת תהליך

□ תהליך אחד (האב) יכול ליצור תהליך אחר (הבן)
■ שדה ppid בביצוע ps -al ב-Linux.

□ בדרך-כלל, האב מגדיר או מוריש משאבים ותכונות לבניו.
■ ב-Linux, הבן יורש את שדה user, ועוד.

□ האב יכול להמתין לבנו, לסיים, או להמשיך לרוץ במקביל.
■ רק תהליך אב יכול להמתין לבניו

ב Windows יש קריאת `CreateProcess(...,prog.exe,...)`
■ מייצרת תהליך חדש (ללא קשרי אבות/בנים) אשר מתחיל לבצע את `prog`.

יצירת תהליכים ב-UNIX: fork()

<pre>int main(int argc, char **argv) { int child_pid = fork(); if (child_pid == 0) { printf("Son of %d is %d\n", getpid(), getppid()); return 0; } else { printf("Father of %d is %d\n", child_pid, getpid()); return 0; } }</pre>	<ul style="list-style-type: none">□ יוצר ומאתחל PCB.□ מיצר מרחב כתובות חדש, ומאתחל אותו עם העתק מלא של מרחב הכתובות של האב.□ מאתחל משאבי גרעין לפי משאבי האב (למשל, קבצים פתוחים)■ באג נפוץ הוא שכיחת סגירת קבצים פתוחים של האב אצל הבן□ שם את ה-PCB בתור המוכנים□ עכשיו יש שני תהליכים, אשר נמצאים באותה נקודה בביצוע אותה תוכנית.□ שני התהליכים חוזרים מה fork: ■ הבן, עם ערך 0 ■ האב, עם מספר התהליך (pid) של הבן <p>נראה בהמשך מימוש יעיל יותר ל fork()</p>
--	---

איך מפעילים תוכנית חדשה?

```
int execv(char *prog, char **argv)
```

- עוצר את ביצוע התוכנית הנוכחית.
- טוען את prog לתוך מרחב הכתובות.
- מאתחל את מצב המעבד, וארגומנטים עבור התוכנית החדשה.
- מפנה את המעבד (ה-PCB מועבר לתור המוכנים).

לא יוצר תהליך חדש!

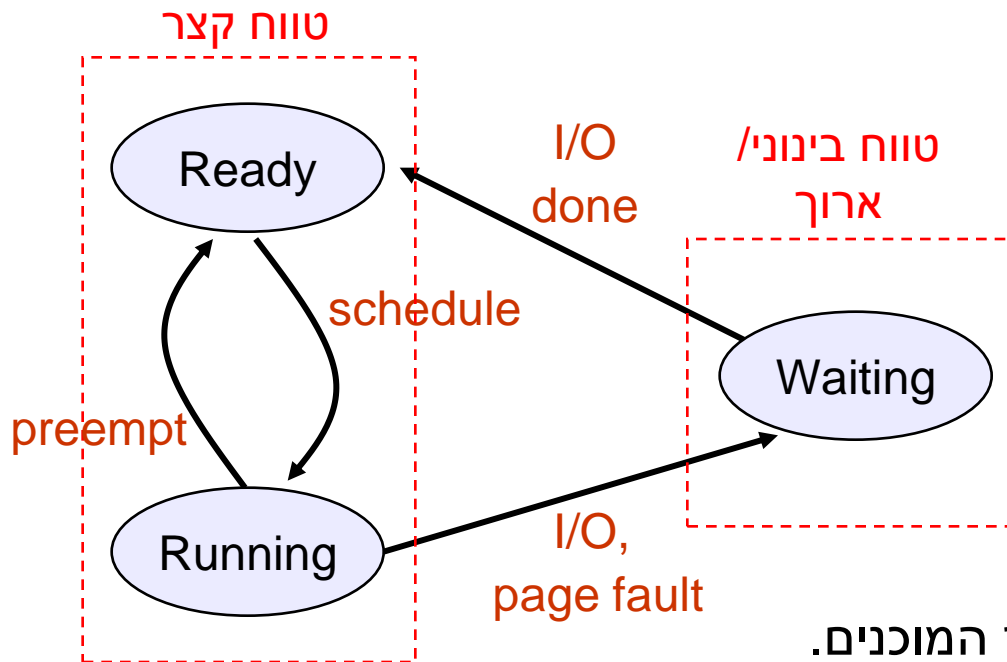
UNIX shell :kNd1?

```
int main(int argc, char **argv)
{
    while (1) {
        char *cmd = get_next_command();
        int child_pid = fork();
        if (child_pid == 0) {
            execv(cmd);
            fprintf(stderr, "exec failed!");
        } else {
            wait(child_pid);
        }
    }
}
```

זימון תהליכים

- מדיניות בסיסיות: RR, FCFS, SJF
- הערכת זמן ריצה ושימוש בעדיפויות
- זימון מבוסס עדיפויות וריבוי תורים
- דוגמאות: Unix, Linux, Windows

לימון תהליכים



□ זימון טווח-קצר:

- בוחר תהליך מתור המוכנים ומריץ אותו ב-CPU.
- מופעל לעיתים קרובות (אלפיות-שנייה).
- חייב להיות מהיר.

□ זימון טווח-ארוך:

- בוחר איזה תהליך יובא לתור המוכנים.
- מופעל לעיתים רחוקות (שניות, דקות).
- יכול להיות איטי.

- תהליכים יכולים להשתחרר **יחד** מהמתנה (למשל, אחרי המתנה ל timer) או **בבודד** (למשל, אחרי המתנה למשאב שאינו ניתן לשיתוף כמו מדפסת)

מדדים להערכת אפלטוןיות לזמן תהליכים

עבור זימון טווח-קצר, המדדים העיקריים הם:

□ **זמן שהייה** מינימאלי (= זמן המתנה + זמן ביצוע)

□ **תקורה** מינימאלית

□ אי-אפשר לנצח בשני המדדים (trade-off). 

לפעמים מעוניינים במטרות נוספות:

□ **ניצול** של המעבד: כמה זמן המעבד פעיל

□ **תפוקה** (throughput): כמה תהליכים מסתיימים בפרק זמן

אף־אוריתם First-Come, First-Served

□ התהליך שהגיע ראשון לתור הממתינים ירוץ ראשון

■ נותן עדיפות לתהליכים **חישוביים** (CPU bound)

■ ממזער ניצול התקנים

■ לא מספק דרישות שיתוף (time sharing)

◀ non-preemptive (ללא הפקעות): תהליך מקבל את המעבד עד לסיומו.

◀ מימוש פשוט: תור התהליכים המוכנים הוא FIFO.

FCFS: קצת נכונה

זמן ההמתנה של תהליך מרגע הגעתו לתור המוכנים ועד לתחילת ביצועו תלוי בסדר הגעת התהליכים לטווח הקצר למשל, אפשרות אחת (התהליכים מגיעים ביחד):



זמן המתנה ממוצע = $17 = (0+24+27)/3$.



זמן המתנה ממוצע = $3 = (0+3+6)/3$.

אפקט השיירה Convoy

C תהליך עתיר חישובים
 I_1, \dots, I_n תהליכים עתירי I/O

מה קורה?

□ תהליך C תופס את המעבד.

■ תהליכי I_j מצטברים בתור המוכנים.

■ התקני קלט / פלט מובטלים!



Round Robin (RR)

- תור מוכנים מעגלי
- המעבד מוקצה לתהליך הראשון בתור
- אם זמן הביצוע של התהליך גדול **מקצבת זמן** מסוימת, q , התהליך מופסק ומועבר לסוף תור המוכנים.
- ◀ preemptive
- ◀ בדרך-כלל, $q = 10-100\text{msec}$.
- ◀ מימוש על-ידי timer שמייצר פסיקה כל q יחידות-זמן.

זמן סיום: RR *f* FCFS

RR	FCFS	תהליך
991	100	1
992	200	2
		⋮ ↓
1000	1000	10

10 תהליכים □

כל תהליך דורש 100 יחידות-זמן. □

$q=1$. □

Shortest Job First (SJF)

נקרא גם Shortest Processing Time First
מריצים את התהליך עם זמן ביצוע מינימאלי, עד לסיומו.

■ כל התהליכים מגיעים יחד.

■ זמן הביצוע של תהליך ידוע מראש.

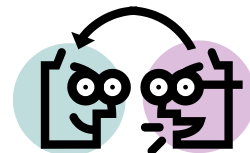
◀ Non-preemptive.

דוגמא: P1 (6) P2 (8) P3 (7) P4 (3)



זמן המתנה ממוצע $= 7 = (0+3+9+16)/4$

מינימאלי לכל סדר זימון אפשרי!



מדדי יציאות למאנאלי לימון

T_i זמן שהייה של תהליך בטווח הקצר (רץ או מוכן)
 t_i זמן הריצה (סכ"ה זמן חישוב) של תהליך

זמן שהייה ממוצע של תהליך, תחת מדיניות זימון A

כאשר N הוא מספר התהליכים

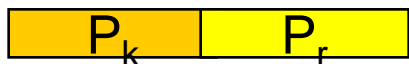
$$H_A = \frac{1}{N} \cdot \sum_{k=1}^N T_k$$

להפקיץ או לא להפקיץ?

למה: לכל מדיניות זימון עם הפקעה A עבור N תהליכים שמגיעים יחד, קיימת מדיניות A' ללא הפקעה כך ש- $H_{A'} \leq H_A$

הוכחה

בהינתן זימון של N תהליכים לפי מדיניות A
יהי P_k התהליך שמסתים אחרון



נצופף את ריצת P_k לסוף הזימון



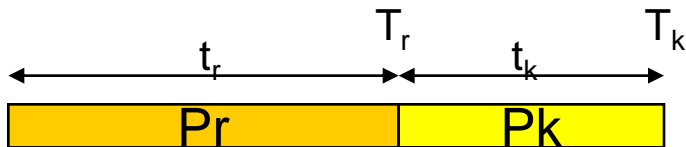
$$\frac{1}{N} \cdot [T'_1 + \dots + T_k + \dots + T'_N]$$

זמן השהייה של P_k לא השתנה
וזמן השהייה של התהליכים האחרים לא גדל
נחזור על הפעולה עבור יתר התהליכים בלי P_k

זמן שהייה מחוצ אופטימלי SJF

משפט: כשתהליכים מגיעים יחד, לכל מדיניות זימון A $H_{SJF} \leq H_A$
הוכחה

לפי הלמה, ניתן להניח ש- A היא מדיניות ללא הפקעה.



אם A לא SJF, יש שני תהליכים כך ש
 P_r מתוזמן מיד לפני P_k , אבל $t_k < t_r$



נחליף בין P_k ו P_r

זמן שהייה ממוצע אחרי ההחלפה לא גדל:

$$\frac{1}{N} \cdot [T_1 + \dots + (T_r + t_k) + \dots + (T_k - t_r) + \dots + T_N] = \frac{1}{N} \cdot [\sum T_i + (t_k - t_r)]$$

למן שהייה ממוצעת תחת RR

משפט: כשתהליכים מגיעים יחד, $H_{RR} \leq 2H_{SJF}$

הוכחה: נסמן ב- $\text{delay}_A(i, j)$ את העיכוב שנגרם לתהליך P_j בגלל שתהליך P_i רץ (במדיניות זימון A).

תמיד

$$N \cdot H_A = \sum_{k=1}^N T_k = \sum_{i=1}^N \sum_{j=1}^N \text{delay}_A(i, j)$$

$$= \sum_{i=1}^N t_i + \sum_{1 \leq i < j \leq N} [\text{delay}_A(i, j) + \text{delay}_A(j, i)]$$

למן שהייה ממוצעת תחת RR

משפט: כשתהליכים מגיעים יחד, $H_{RR} \leq 2H_{SJF}$

הוכחה: נסמן ב- $\text{delay}_A(i, j)$ את העיכוב שנגרם לתהליך P_j בגלל שתהליך P_i רץ (במדיניות זימון A).

$$N \cdot H_A = \sum_{i=1}^N t_i + \sum_{1 \leq i < j \leq N} [\text{delay}_A(i, j) + \text{delay}_A(j, i)] \quad \text{תמיד}$$

$$N \cdot H_{SJF} = \sum_{i=1}^N t_i + \sum_{1 \leq i < j \leq N} \min(t_i, t_j) \quad \text{עבור SJF}$$



זמן שהייה ממוצע תחת RR

משפט: כשתהליכים מגיעים יחד, $H_{RR} \leq 2H_{SJF}$

הוכחה: נסמן ב- $\text{delay}_A(i, j)$ את העיכוב שנגרם לתהליך P_j בגלל שתהליך P_i רץ (במדיניות זימון A).

$$N \cdot H_A = \sum_{i=1}^N t_i + \sum_{1 \leq i < j \leq N} [\text{delay}_A(i, j) + \text{delay}_A(j, i)] \quad \text{תמיד}$$

$$N \cdot H_{SJF} = \sum_{i=1}^N t_i + \sum_{1 \leq i < j \leq N} \min(t_i, t_j) \quad \text{עבור SJF}$$

$$N \cdot H_{RR} = \sum_{i=1}^N t_i + \sum_{1 \leq i < j \leq N} 2 \cdot \min(t_i, t_j) \quad \text{עבור RR}$$



למן שהייה ממוצעת תחת RR

משפט: כשתהליכים מגיעים יחד, $H_{RR} \leq 2H_{SJF}$

הוכחה: נסמן ב- $\text{delay}_A(i, j)$ את העיכוב שנגרם לתהליך P_j בגלל שתהליך P_i רץ (במדיניות זימון A).

$$N \cdot H_A = \sum_{i=1}^N t_i + \sum_{1 \leq i < j \leq N} [\text{delay}_A(i, j) + \text{delay}_A(j, i)] \quad \text{תמיד}$$

$$N \cdot H_{SJF} = \sum_{i=1}^N t_i + \sum_{1 \leq i < j \leq N} \min(t_i, t_j) \quad \text{עבור SJF}$$

$$N \cdot H_{RR} = \sum_{i=1}^N t_i + \sum_{1 \leq i < j \leq N} 2 \cdot \min(t_i, t_j) \quad \text{עבור RR}$$

Shortest Remaining Time to Completion First (SRTF)

כאשר מגיע תהליך P_i שזמן הביצוע הנותר שלו קצר יותר
מזמן הביצוע הנותר של התהליך שרץ כרגע P_k
מכניסים את P_i למעבד, במקום P_k

preemptive 

ממזער את זמן השהייה הממוצע במערכת. 

SRTF אופטימאלי כאשר תהליכים לא מגיעים יחד, בניגוד
ל-SJF.

ניבוי זמן הריצה של תהליך

אומדן סטטיסטי של הזמן עד לויתור על המעבד, על-פי הפעמים הקודמות שהתהליך החזיק במעבד

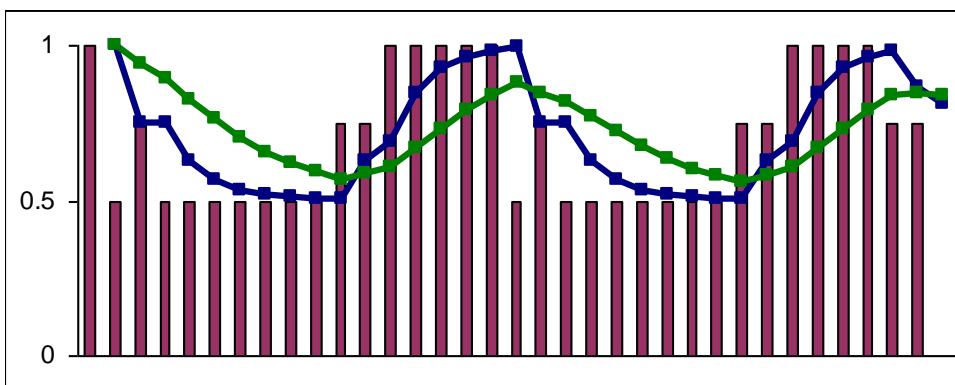


■ τ_i – הערכת זמן הביצוע לסיבוב ה- i

■ t_i – זמן הביצוע בפועל בסיבוב ה- i

$$\tau_{i+1} = \alpha \cdot \tau_i + (1 - \alpha) \cdot t_i$$

$$0 \leq \alpha \leq 1$$



לדוגמא,

$$t_0 = 1$$

$$\alpha = 1/2$$

$$\alpha = 3/4$$

אחת הגישות היא ניבוי הזמן הדרוש לפעולה הבאה, בהסתמך על היסטוריה של התהליך

נסמן ב- $T(i)$ את זמן הפעולה האינטראקטיבית ה- I - ית שלקח לבצעה בפועל

נסמן ב- $E(i)$ מהו הניחוש לזמן הפעולה האינטראקטיבית ה- I - ית.

הניחוש ההתחלתי למשך הזמן של הפעולה האינטראקטיבית הראשונה צריך להיות נתון $E(1)$

הניבוי לזמן הפעולה האינטראקטיבית הבאה מחושב לפי הנוסחה:

$$E(i+1) = a T(i) + (1-a) E(i) \quad \text{כאשר } 0 < a < 1$$

הנוסחה מבצעת שקלול של זמני הפעולה שהיו בעבר. על ידי בחירת ערכו של a אפשר לייחס

חשיבות רבה או פחותה להיסטוריה. למשל, על ידי בחירת $a = 0.5$ נקבל את סדרת הניבויים

$$E(1)$$

$$E(2) = 0.5 T(1) + 0.5 E(1)$$

$$E(3) = 0.5 T(2) + 0.25 T(1) + 0.25 E(1)$$

$$E(4) = 0.5 T(3) + 0.25 T(2) + 0.125 T(1) + 0.125 E(1)$$

זימון לפי עדיפויות



□ לכל תהליך יש עדיפות התחלתית

□ עדיפות התחלתית גבוהה ניתנת



■ לתהליכים שסיומם דחוף



■ לתהליכים אינטראקטיביים

□ התהליך עם העדיפות הגבוהה ביותר מקבל את המעבד.

■ SJF הוא זימון לפי עדיפויות כאשר העדיפות היא ההופכי של זמן הביצוע.

הרעבה של תהליכים עם עדיפות נמוכה 

הזדקנות (זמן שהייה ארוך) של תהליכים גורמת להגדלת העדיפות שלהם (לדוגמא, selfish round-robin). 

Multilevel Feedback Queues



- קיימים מספר תורים לפי סדר עדיפות.
- תור גבוה לתהליכים בעלי עדיפות גבוהה יותר
- לתורים הנמוכים מוקצה אחוז קטן יותר של זמן מעבד.
- quantum גדול יותר לתורים נמוכים יותר.
- תהליך מתחיל בתור הגבוה ביותר.
- בסוף ה-quantum, יורד לתור נמוך יותר.
- **אפליה מתקנת** לתהליכים עתירי ק/פ לעומת תהליכים עתירי-חישוב
- תהליך שמשחרר את המעבד לפני סוף ה-quantum (בגלל פעולת קלט / פלט), חוזר לתור גבוה יותר.
- **עדיפויות דינמיות** לפי השימוש במעבד (+ עדיפות התחלתית):
- המתנה ארוכה למעבד - מגדילה את העדיפות
- ריצה ארוכה במעבד - מקטינה את העדיפות

Lottery scheduling

הפעלת שיטת התזמון המובטח יכולה לדרוש תקורה לא קטנה. למשל, במקרה של ההבטחה על צריך לחשב מחדש את היחס לכל תהליכי המערכת כאשר מגיע, חלוקה שווה של זמני CPU למערכת תהליך חדש או כאשר אחד מהתהליכים עוזב את המערכת. שיטת ההגרלה יכולה להפחית את התקורה של חישוב המידות המובטחות ולהביא לתוצאות דומות באופן סטטיסטי.

מתזמן ייתן לכל תהליך חדש שנכנס למערכת כרטיס, במקרה של חלוקה שווה של זמני CPU החלטות התזמון מתבצעות בפרקי זמן שמשכם נגזר. הגרלה המזכה בנתח כלשהו של זמן CPU מכמות הכרטיסים המשתתפים בהגרלה או מקריאת מערכת והגעת תהליך חדש. בעת החלטת כך שלאורך זמן השיטה. תזמון מתבצעת הגרלה שבעקבותיה נבחר תהליך שיקבל את ה-CPU משיגה חלוקה שווה למדי של זמני העיבוד. בכל החלטת השיטה איננה גורמת לתקורה, כי אין צורך לחשב את היחס של שימוש ה-CPU תזמון. כל מה שיש הוא לעדכן את משך הזמן של הריצה הבאה.

לימון ק-UNIX

□ זימון לפי עדיפויות

□ עדיפות מספרית נמוכה = עדיפות טובה יותר

□ חישוב העדיפות מתבסס על דעיכה אקספוננציאלית

■ תהליך שהשתמש לא מזמן במעבד מקבל עדיפות גבוהה (גרועה)

■ ככל שעובר הזמן, עדיפותו של התהליך קטנה (משתפרת)

⇐ תהליכים שיוותרו מרצונם על המעבד (עתירי ק / פ) יחזרו אליו מהר יותר מתהליכים שעברו הפקעה (עתירי חישוב)

חיסוק עציוניות ב-UNIX

עדיפות תהליך j בתחילת יחידת הזמן ה- t היא:

$$P_j(t) = Base_j + NICE_j + \frac{CPU_j(t-1)}{2}$$

$$CPU_j(t) = \frac{1}{2}U_j(t) + \frac{1}{2}CPU_j(t-1)$$

$Base_j$ = Base priority of Process j

$NICE_j$ = User controllable adjustment factor

$CPU_j(t)$ = Average processor utilization by j

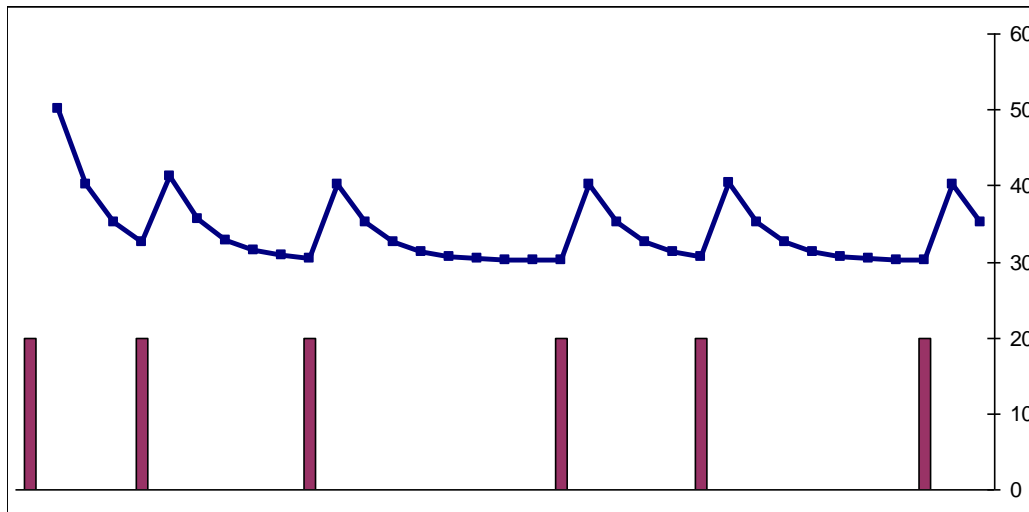
$U_j(t)$ = Processor utilization of j in interval i

חיסוק עציוניות ב-UNIX

עדיפות תהליך j בתחילת יחידת הזמן ה- i היא:

$$P_j(t) = Base_j + NICE_j + \frac{CPU_j(t-1)}{2}$$

$$CPU_j(t) = \frac{1}{2}U_j(t) + \frac{1}{2}CPU_j(t-1)$$



$Base = 40$

$NICE = -10$



ליאון ב Linux: רציון בסיסי

- גרסה נוספת של תורי עדיפויות
- נדון רק בזימון תהליכים רגילים
- זימון תהליכי זמן-אמת (real time) יתואר בתרגול.
- בדומה ל-unix, לכל תהליך יש עדיפות המורכבת מערך בסיס קבוע + בונוס דינמי
- המתכנת יכול לשנות את ערך הבסיס באמצעות קריאת המערכת `nice()`
- הבונוס הדינמי גדל כשתהליך חוזר מהמתנה וקטן (עד לערך שלילי) כאשר התהליך נמצא הרבה בטווח הקצר
- ⇐ תהליך עתיר ק/פ צפוי לקבל עדיפות גבוהה יותר מתהליך עתיר חישוב בעל אותה עדיפות בסיס



לימון ב Linux: מנות זמן

- תהליך מקבל time slice שאורכו תלוי בעדיפות הבסיס
 - תהליך עדיף מקבל time slice ארוך יותר.
 - תהליך משובץ לתור עדיפות בהתאם לערך הנוכחי של עדיפותו הכוללת
 - זמן המעבד מחולק ל**תקופות** (epoch)
 - בכל תקופה, המערכת נותנת לכל התהליכים לרוץ, החל מאלו הנמצאים בתור העדיפות הגבוה ביותר
 - כל תהליך רץ עד סיום ה-time slice שלו
- ואז מקבל time slice חדש עבור התקופה הבאה

ליאון ב Linux: תהליכים חישוביים ואינטראקטיביים



- תהליכים **חישוביים** רוצים הרבה זמן מעבד, אבל יכולים לחכות
- תהליכים **אינטראקטיביים** רוצים מעט זמן מעבד, אבל מייד
- מאפיינים תהליך כאינטראקטיבי אם ממתין הרבה זמן מיוזמתו (בטווח הבינוני), כחלק מזמן הריצה הכולל שלו
- רוצים לתת לתהליכים אינטראקטיביים כמה זמן שנחוץ
- מחדשים את ה time slice לריצה נוספת בתקופה הנוכחית



עוד מידע בתרגול!



זיאון ק-NT Windows

□ גם כאן, שימוש בתורים לפי עדיפות

■ 32 תורים (dispatcher ready queues)

□ Real time priority (עדיפויות 16-31)

□ Variable priority (עדיפויות 1-15)

□ (עדיפות 0 שמורה למערכת)

□ מדיניות Round Robin

■ על התור העדיף ביותר שאינו ריק

□ העדיפות יורדת אם נצרך כל ה-quantum

□ העדיפות עולה אם תהליך עובר מ-wait ל-ready

■ העדיפות נקבעת על-פי סוג הקלט / פלט

□ ק"פ מהמקלדת מקנה עדיפות גבוהה

תהליכים-זייאט: חוטים

- מוטיבציה
- חוטי משתמש וחוטי מערכת
- תמיכת מערכת ההפעלה
- דוגמאות ושימושים

צלות ריבוי תהליכים

□ תהליכים דורשים משאבי מערכת רבים

■ מרחב כתובות, גישה לקלט/פלט (file table)...

□ זימון תהליכים הינו פעולה כבדה

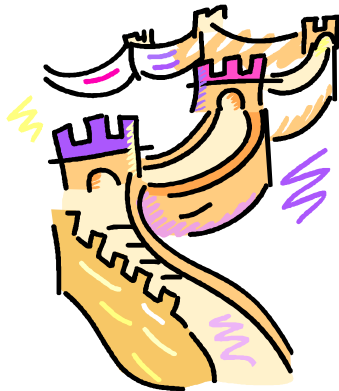
■ context switch לוקח הרבה זמן.

□ תקשורת בין תהליכים עוברת

דרך מערכת ההפעלה

■ מערכת ההפעלה שומרת על הגבולות

בין תהליכים שונים



טכניקות אקדמיות

... אבל במקרים רבים קל יותר לפתור בעיות באמצעות ריבוי תהליכים

דוגמא 1: מעבד תמלילים

- מציג פלט, ממתין לקלט מהמשתמש, בודק איות, ...

דוגמא 2: שרת קבצים / דואר

- ממתין לקבל בקשה

- כאשר מקבל בקשה, חייב להפסיק לחכות ולעבור לטפל בבקשה

פתרון:

- לבצע fork

- תהליך הבן מטפל בבקשה שהגיעה

- האב ממשיך להמתין לבקשות חדשות

זה פתרון לא-יעיל כי מחייב הקצאת מרחב כתובות, PCB ...

שוארי הפקד

□ התהליכים שנוצרים לטיפול בבקשות דומים זה לזה:

■ אותו קוד

■ אותם משאבים ונתונים

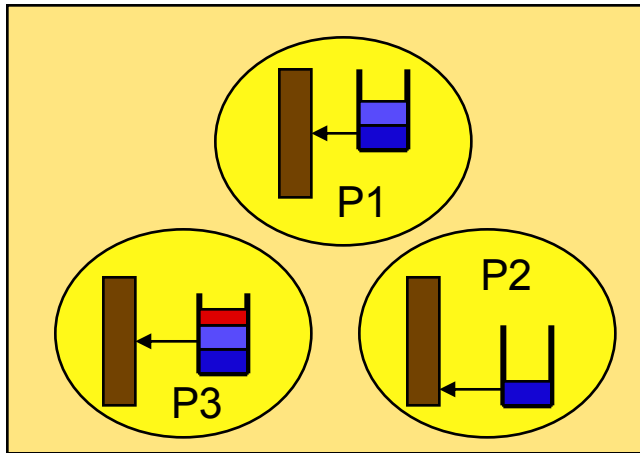
□ אבל לא זהים:

■ מטפלים בבקשות שונות

■ נמצאים בשלבים שונים במהלך הטיפול

רעיון! תהליכים-דייט (lightweight processes) אשר
משתפים מרחב כתובות, הרשאות ומשאבים

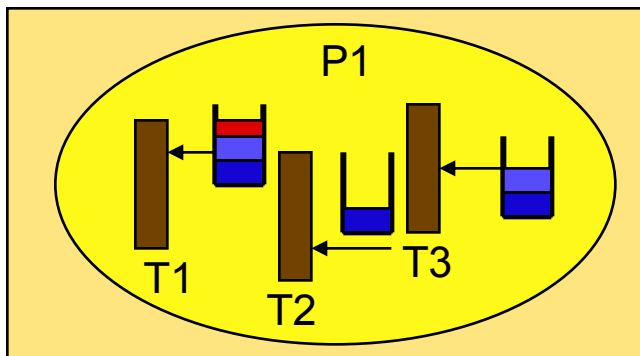
תהליכים-דיואט = חוטים



□ חוט (thread) הינו יחידת ביצוע (בקרה) בתוך תהליך

במערכות הפעלה קלאסיות
"חוט" יחיד בכל תהליך

במערכות הפעלה מודרניות
תהליך הוא רק מיכל לחוטים



□ לכל חוט מוקצים המשאבים הבאים:

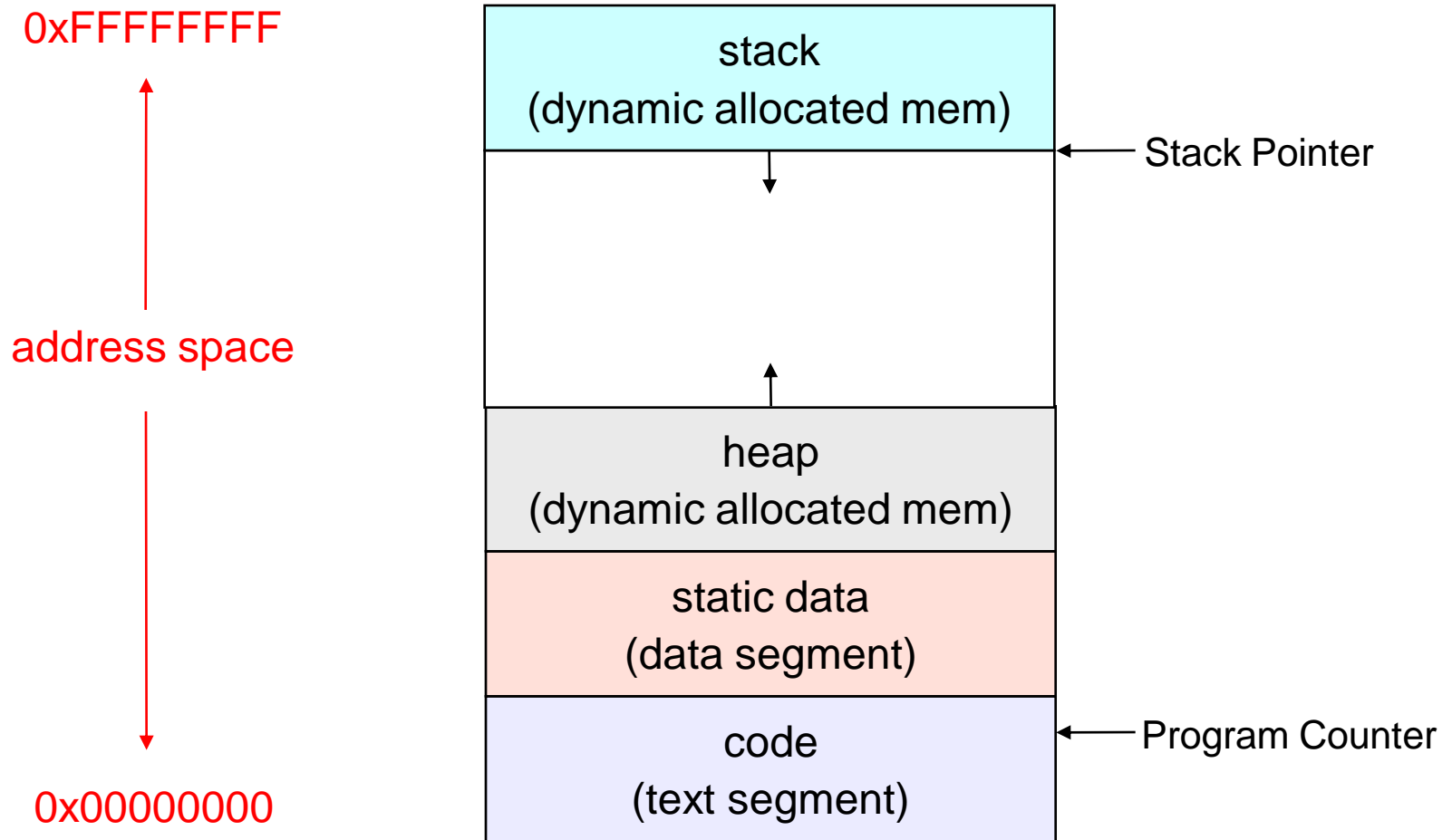
program counter ■

מחסנית ■

רגיסטרים ■

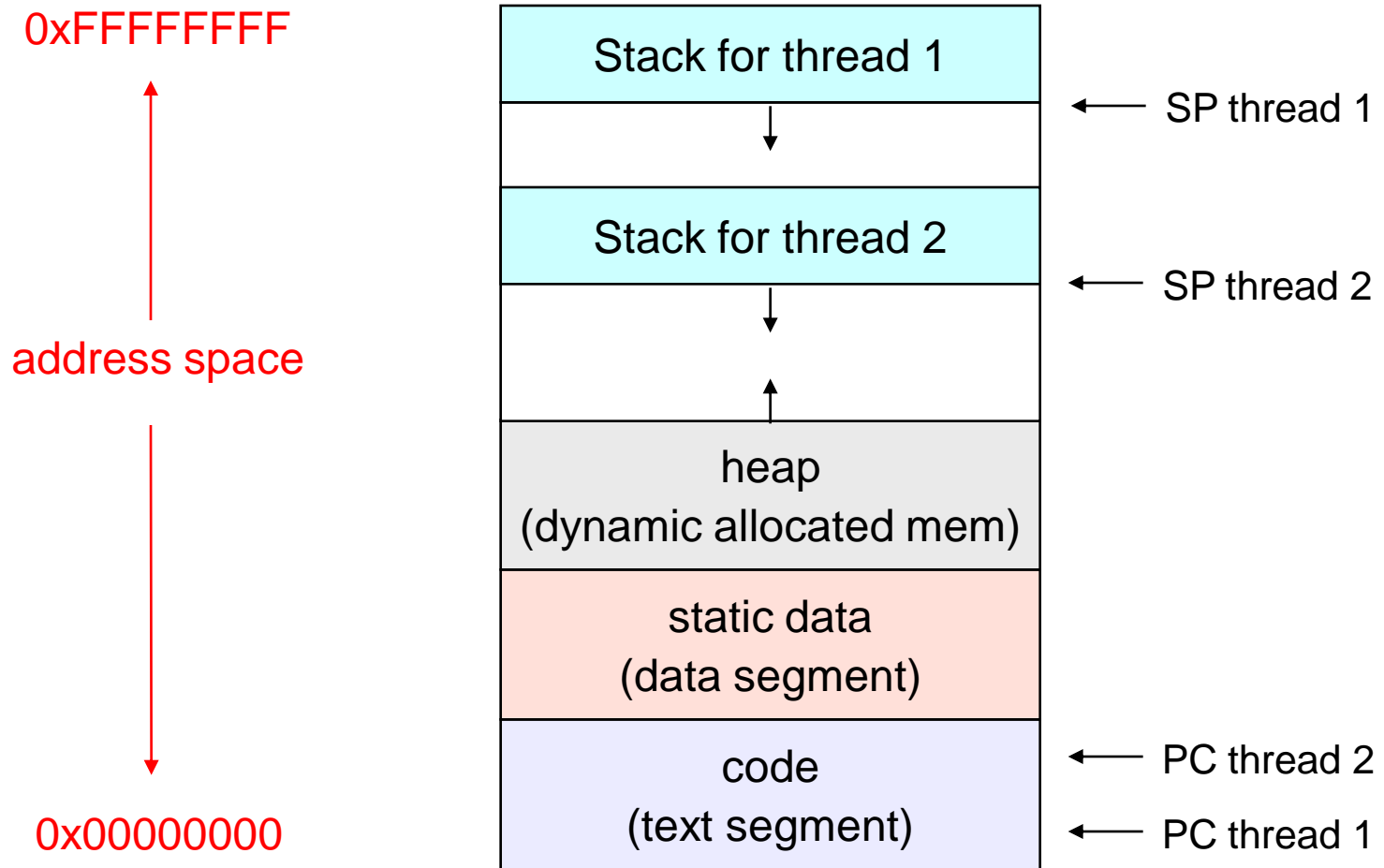
נמצאים ב Thread Control Block (TCB)

תזכורת: מרחב הכתובות של תהליך



מרחב הכתובות של תהליך

מנוקה-חוסים



חוסים עצומת תהליכים

ייחודי לתהליך ייחודי לחוט

✓

✓

Program Counter □

✓

✓

Registers □

✓

✓

Execution Stack □

x

✓

Address Space □

x

✓

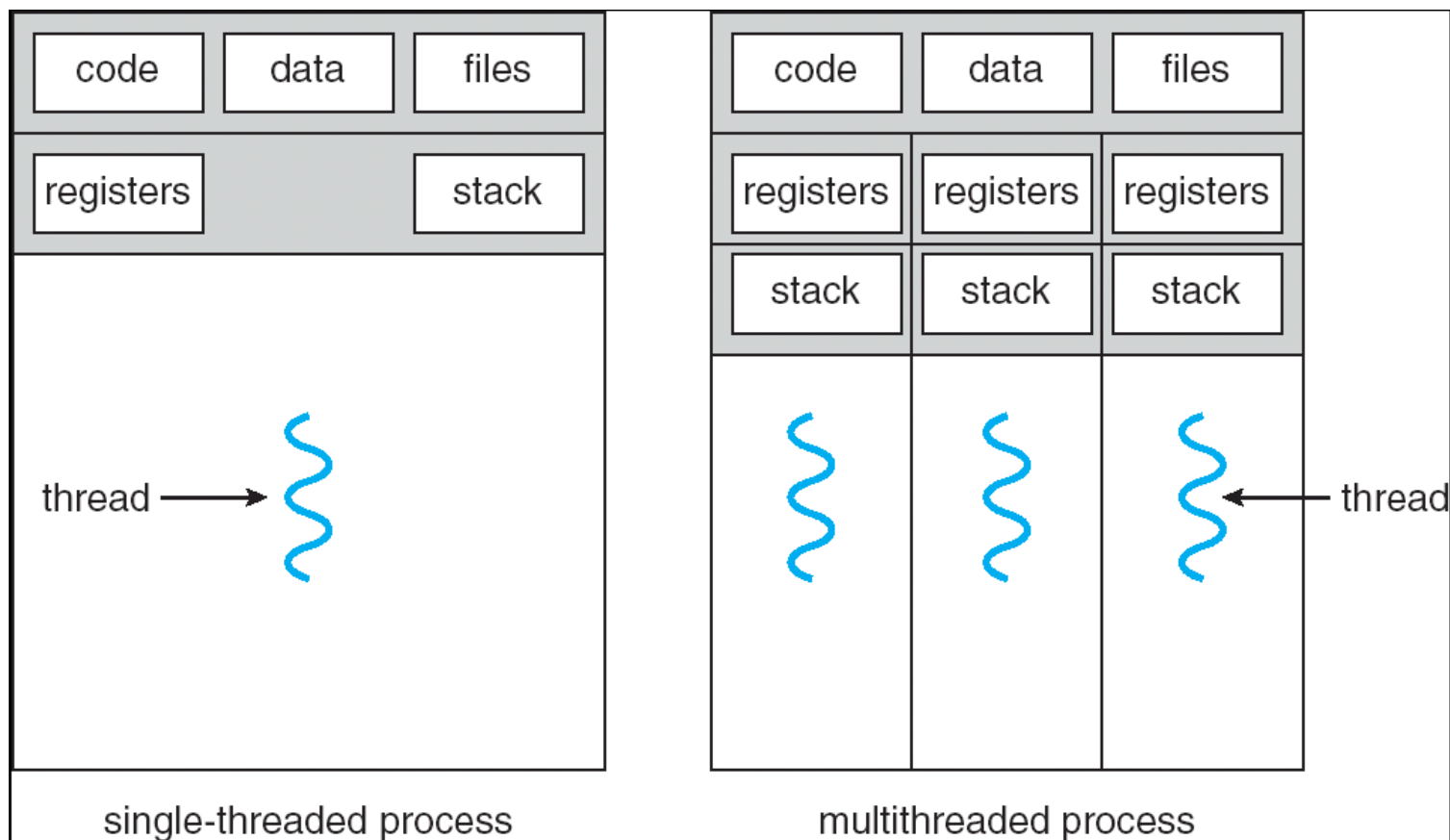
Open Files □

x

✓

File position □

תהליך מרובה חוטים: תרשים



דוא"ל: שרת קהלים

□ תהליך מקבל בקשות עם

הפרמטרים הבאים:

■ סוג: קריאה / כתיבה

■ זיהוי קובץ

■ מיקום בקובץ

■ אורך

■ חוצץ (buffer)

□ התהליך מחזיר:

■ מחרוזת תווים (במקרה של

קריאה)

■ סטאטוס (הצלחה/כישלון)

■ אורך קריאה/כתיבה

(בפועל)

שרת קבצים: מ'מ'מ' עם חוט יחיד

```
do forever
  get request;
  execute request;
  return results;
end;
```

✓ פשטות

✗ ניצול משאבים לקוי, למשל בהמתנה לקלט/פלט

ניתן ליעל ע"י קלט / פלט אסינכרוני וטיפול במספר בקשות
בו זמנית

מימוש שרת באמצעות חוטים

□ חוט מנהל

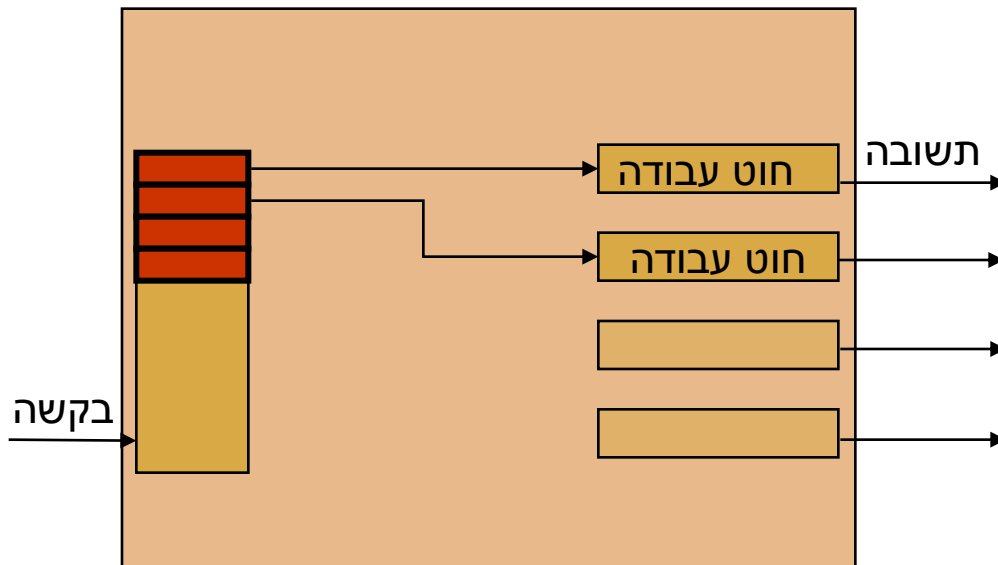
- מקבל בקשה

- מייצר חוט עבודה, ומעביר אליו את הבקשה

□ חוט עבודה

- מבצע את הבקשה

- מחזיר תשובה



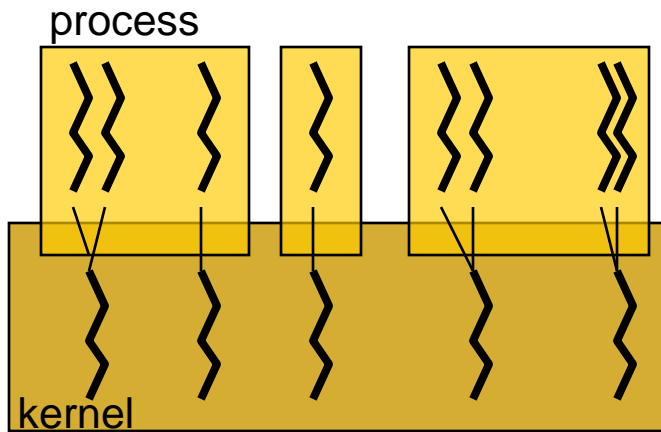
- ✓ תפוקה גבוהה יותר

- ✗ תכנות מורכב יותר

יתרונות וחסרונות

- ✓ יצירת חוט יעילה יותר
 - רק יצירת thread control block והקצאת מחסנית
 - החלפת הקשר בין חוטים של אותו תהליך מהירה יותר
- ✓ ניצול טוב יותר של משאבים
 - חוט אחד נחסם (למשל על IO), חוטים אחרים של אותו תהליך ממשיכים לרוץ
 - מקביליות אמיתית במערכות מרובות מעבדים
- ✓ תקשורת נוחה יותר בין חוטים השייכים לאותו תהליך
 - זיכרון משותף
- ✓ תכנות מובנה יותר
- ✗ חוסר הגנה בין חוטים באותו תהליך
 - חוט עלול לדרוס את המחסנית של חוט אחר
 - גישה לא מתואמת למשתנים גלובליים

חוטי משתמש וחוטי מערכת



□ חוט משתמש (user threads)

- מוגדרים ע"י סביבת התכנות
- לא דורשים קריאות מערכת
- זימון בשיתוף פעולה (ע"י פקודת yield)
- אין החלפת הקשר בגרעין
- מה קורה כאשר חוט נחסם?

□ חוט מערכת (kernel threads)

- מוכרים למערכת ההפעלה
- נקראים lightweight processes

חוטי משתמש מאפשרים לאפליקציה לחקות ריבוי חוטים גם במערכת הפעלה single-threaded.

תמיכת אצרכת הפצה בחוטים

□ יצירת והריסת חוטים, לדוגמא

```
thread_create(char *stack)
thread_exit(...)
thread_join(...)
thread_kill(...)
```

□ מבני נתונים פנימיים

thread control block ■

Program counter, stack, registers ■

□ מנגנוני סנכרון

■ למשל, תיאום גישה לזיכרון משותף

POSIX חוטי: חוטי

□ ממשק סטנדרטי (IEEE 1003.1c) ליצירת וניהול חוטים.

■ רק Application Program Interface.

□ דומה מאוד לפעולות המתאימות עבור ניהול תהליכים:

pthread_create

pthread_exit

pthread_join

□ כולל מנגנונים רבים לתיאום בין חוטים.

□ נתמך בהרבה מערכות "UNIX"

■ בפרט, NPTL ב Linux (ולפניה PThreads)

■ מימושים שונים

□ נתמך גם ב Windows

השוואת ביצועים Pthreads

יחס	זמן יצירה / סיום:		
56	251	fork / exit	תהליכים
21	94	pthread_create / pthread_join	חוטי גרעין
—	4.5	pthread_create / pthread_join	חוטי משתמש

במיקרו שניות, על 700MHz Pentium, עם Linux 2.2.16
[Steve Gribble, 2001]

Solaris *השוואת ביצועים*

יחס	זמן יצירה	
33	1700	תהליכים
7	350	חוטי גרעין
—	52	חוטי משתמש

במיקרו שניות, על SPARCstation2 (Sun 4/75).

[Solaris Multithreading Guide]

<http://docs.sun.com/app/docs/doc/801-6659>

Windows NT :אנדל

- היחידה הבסיסית הינה תהליך.
- תהליך יכול להכיל כמה חוטים – kernel threads.
- חוט יכול להכיל כמה סיבים (fibers),
שהם בעצם user threads
- זימון נעשה ברמת החוטים
- תהליך רק מגדיר את מרחב הזיכרון, ומהווה מיכל לחוטים

כמה חוטים ליצור בשרת?

אם כל חוט צריך t_c יחידות זמן ולתהליך מוקצים t_p יחידות זמן (בכל שניה). כמה חוטים כדאי לתהליך לייצר?

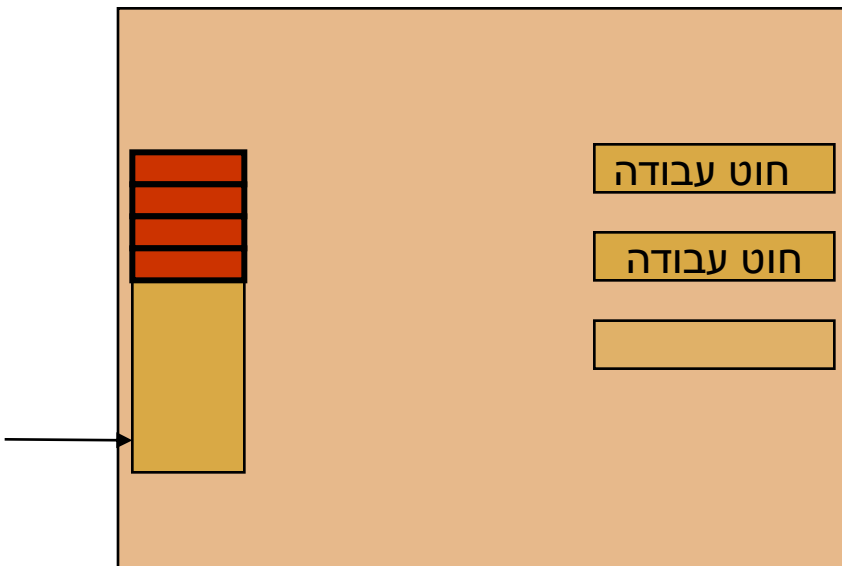
אין טעם לייצר יותר מ- t_p / t_c חוטים. 

■ ניצול מלא של זמן המעבד המוקצה לתהליך

■ פחות תקורה להחלפת הקשר

■ פחות פעולות סינכרון

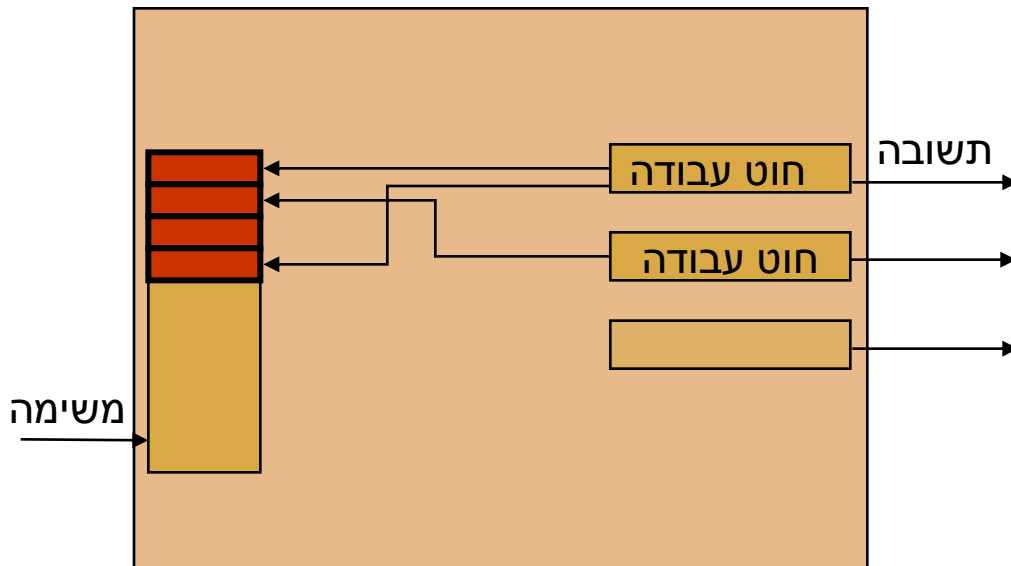
■ יותר מחסנית לכל חוט



thread pooling מאגר חוטים

- יוצרים מספר קבוע של חוטי עבודה ראשי אחד
- החוט הראשי מאכסן את הבקשות המגיעות בתור FIFO

- חוטי העבודה מריצים את הקוד הבא בלולאה אינסופית:



- קח משימה מהתור
- בצע את המשימה
- שלח תשובה ללקוח

(כל חוטי העבודה במקביל)

מאגר חוטים באודף דינאמי

□ t_p ו t_c אינם ידועים, ואולי אפילו אינם קבועים

□ ערכו של t_c עשוי לקבוע את t_p

pool דינאמי 📖

■ מתחילים עם מספר חוטים קבוע

■ אם אורך תור המשימות גדל והתהליך לא מנצל את כל מנת הזמן שלו, מגדילים את מספר החוטים

□ תוספת קבועה או כפלית

□ עד לסף עליון קבוע

■ אם יש הרבה חוטים מובטלים, ניתן לבטל חוטים

תיאום בין תהליכים: יסודות

- דוגמאות לבעיות תיאום
- הגדרות: קטע קריטי, מנעולים
- אלגוריתם קופת-חולים

תיאום

□ תהליכים משתפים פעולה:

- גישה למשאבים משותפים, למשל זיכרון משותף (בעיקר חוטים).
- העברת נתונים מתהליך אחד לשני דרך התקן משותף.

□ חייבים לתאם את השיתוף:

- מניחים שביצוע התהליכים משולב באופן שרירותי. למתכנת האפליקציה אין שליטה על זימון התהליכים.
- שימוש במנגנוני תיאום (synchronization).

דוגמא: בנק הפועלים

□ מימשנו פונקציה למשיכת כסף מחשבון בנק.

```
int withdraw( account, amount) {  
    balance = get_balance( account );  
    balance -= amount;  
    put_balance( account, balance);  
    return balance;  
}
```

□ בחשבון יש \$50000, ושני בעלי החשבון ניגשים
לכספומטים שונים ומושכים \$30000 ו-\$20000 בו-
זמנית.

בנק הפואצ'ים: אין כסף?

□ תהליך נפרד מבצע כל פעולת משיכה (על אותו מעבד)

```
balance = get_balance(account);  
balance -= amount; // 50K-30K  
put_balance(account, balance);  
return balance;    // = 20K
```

```
balance = get_balance(account);  
balance -= amount; // 20K-20K  
put_balance(account, balance);  
return balance;    // = 0
```



בחשבון \$0....

בנק הפואצ'ים: יש כסף!

□ תהליך נפרד מבצע כל פעולת משיכה (על אותו מעבד).

```
balance = get_balance(account);  
balance -= amount; // 50K-30K
```

```
put_balance(account, balance);  
return balance; // = 20K
```

```
balance = get_balance(account);  
balance -= amount; // 50K-20K  
put_balance(account, balance);  
return balance; // = 30K
```



מי שמח עכשיו?

כנח 718

shared in, out

```
procedure echo();  
    read( in, keyboard);  
    out = in;  
→ write( out , screen);  
end echo
```

שני חוטים מבצעים אותו קוד

החלפת חוטים בכל מקום

race condition

תוצאת הריצה אינה צפויה.

עוד דואל: יותר מידי חלבי!

שעה נינט:

יודה:

3:00 מסתכלת במקרר

3:05 הולכת לסופר מסתכל במקרר

3:10 קונה חלב הולך לסופר

3:15 חוזרת הביתה קונה חלב

3:20 מכניסה חלב למקרר חוזר הביתה

3:25 מכניס חלב למקרר



פיתרון 1

להשאיר פתק לפני שהולכים לסופר

```
if (no milk) then
  if (no note) then
    leave note
  buy milk
  remove note
```

קצ'ה עם פתרון 1

```
if (no milk) then  
  if (no note) then
```

```
if (no milk) then  
  if (no note) then  
    leave note  
    buy milk  
    remove note
```

```
leave note  
buy milk  
remove note
```

פעמיים חלב!

פיתרון 2

משאירים פתק לפני שבודקים את המקרר:

Thread A:

```
leave note A
if (no note B) then
  if (no milk) then
    buy milk
remove note A
```

Thread B:

```
leave note B
if (no note A) then
  if (no milk) then
    buy milk
remove note B
```

פעולה עם פתרון 2

leave note A

if (no note B) then

remove note A

leave note B

if (no note A) then
remove note B

אין חלב!

פיתרון 3 😊

לא סימטרי

Thread A:

```
leave note A
while (note B) do nop
if (no milk) then
    buy milk
remove note A
```

Thread B:

```
leave note B
if (no note A) then
    if (no milk) then
        buy milk
remove note B
```

אם שניהם משאירים פתק בו זמנית (race condition), A יקנה חלב!
■ לא הוגן

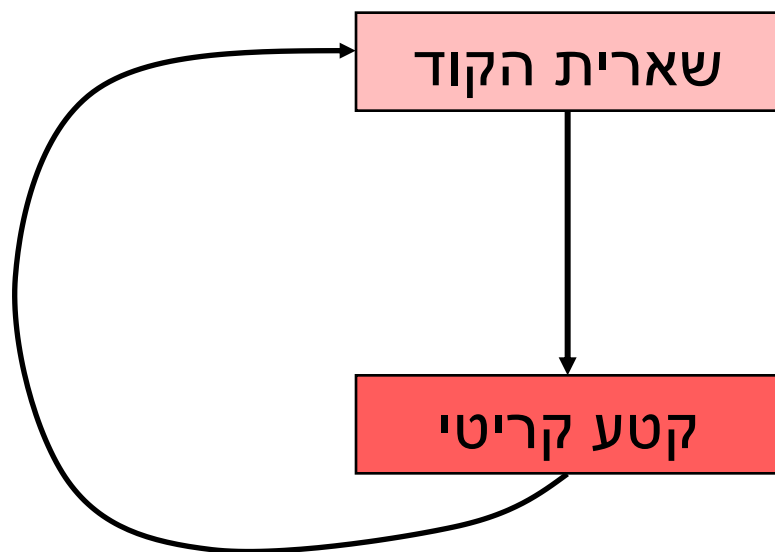
רק לשני תהליכים 😞

לב הכעיה

- שני תהליכים נגשים בו-זמנית לאותו משאב, ללא תיאום.
 - למשל, בגישה למשתנים גלובליים.
 - התוצאה אינה צפויה.
- מנגנון לשליטה בגישות מקביליות למשאבים משותפים.
 - כך שנוכל לחזות באופן דטרמיניסטי את התוצאות.
- לכל מבנה נתונים של מערכת הפעלה (וגם להרבה תוכניות משתמש מרובות-חוטים).
 - Buffers, queues, lists, hash tables

קטע קריטי

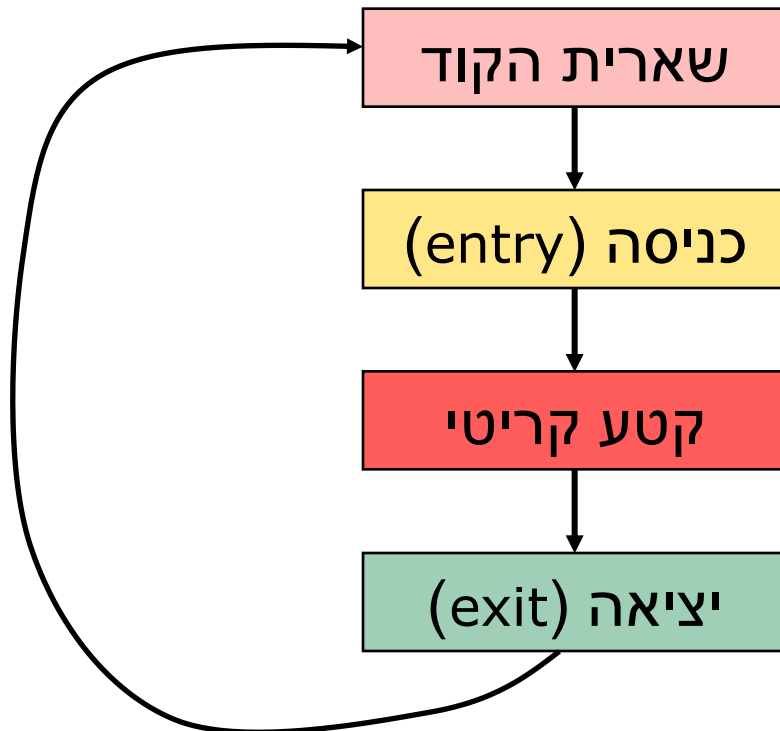
הקוד שניגש למשאב המשותף מכונה **קטע קריטי**



שימו לב: לא בהכרח אותו קוד לכל החוטים
■ חוט אחד מגדיל מונה וחוט שני מקטין אותו.

קטע קריטי

הקוד שניגש למשאב המשותף מכונה **קטע קריטי**
עוטפים אותו בקוד **כניסה** וקוד **יציאה** (של הקטע הקריטי).



אטומיות (atomicity):
בביצוע סדרת פקודות ע"י חוט,
חוטים אחרים אינם רואים
תוצאות חלקיות

באלגוריתם פקציות החלפה

Thread A:

```
leave note A
while (note B) do nop
if (no milk) then
    buy milk
remove note A
```

קוד כניסה

קטע קריטי

קוד יציאה

Thread B:

```
leave note B
if (no note A) then
    if (no milk) then
        buy milk
remove note B
```

Peterson's Solution

Two process solution

The two processes share two variables: int turn; Boolean flag[2]

The variable turn indicates whose turn it is to enter the critical section.

The flag array is used to indicate if a process is ready to enter the critical section. flag[i] = true implies that process P_i is ready.

i

```
flag[i] = TRUE; turn = j;
```

```
while ( flag[j] && turn == j) do noop;
```

```
    CRITICAL SECTION
```

```
    flag[i] = FALSE;
```

```
    REMAINDER SECTION
```

j

```
flag[j] = TRUE; turn = i;
```

```
while ( flag[i] && turn == i) do noop;
```

```
    CRITICAL SECTION
```

```
    flag[j] = FALSE;
```

```
    REMAINDER SECTION
```

תכונות רצויות

מניעה הדדית: חוטים לא מבצעים בו-זמנית את הקטע הקריטי. (mutual exclusion)

- חוטים מעוניינים מחכים בקטע הכניסה.
- כאשר החוט הנוכחי יוצא מהקטע הקריטי, יכולים להיכנס.

תכונות רצויות

מניעה הדדית: חוטים לא מבצעים בו-זמנית את הקטע הקריטי. (mutual exclusion)

התקדמות: אם יש חוטים שרוצים לבצע את הקטע הקריטי, חוט כלשהו יצליח להיכנס. (no deadlock, היעדר קיפאון)
■ אלא אם חוט אחר נמצא בתוך הקטע הקריטי.

תכונות רצויות

מניעה הדדית: חוטים לא מבצעים בו-זמנית את הקטע הקריטי. (mutual exclusion)

התקדמות: אם יש חוטים שרוצים לבצע את הקטע הקריטי, חוט כלשהו יצליח להיכנס. (no deadlock, היעדר קיפאון)

הוגנות: אם יש חוט שרוצה לבצע את הקטע הקריטי, לבסוף יצליח. (no starvation, היעדר הרעבה)

■ רצוי: החוט יכנס לקטע הקריטי תוך מספר צעדים חסום (bounded waiting), ואפילו בסדר הבקשה (FIFO).

מנעולים (locks)

□ **אבסטרקציה** אשר מבטיחה גישה בלעדית למידע באמצעות שתי פונקציות:

- `acquire(lock)` – נחסם בהמתנה עד שמתפנה המנעול.
- `release(lock)` – משחרר את המנעול.

□ `acquire` ו `release` מופיעים בזוגות:

- אחרי `acquire` החוט מחזיק במנעול.
- רק חוט אחד מחזיק את המנעול (בכל נקודת זמן).
- יכול לבצע את הקטע הקריטי.

דואנא פאנאס מאנאזאס

בחזרה לפונקציה למשיכת כסף מחשבון בנק.

```
int withdraw( account, amount) {  
    acquire ( lock ) ;  
    balance = get_balance( account ) ;  
    balance -= amount ;  
    put_balance( account, balance) ;  
    release ( lock ) ;  
    return balance ;  
}
```

} קטע קריטי

המשק הדואל

```
acquire(lock);  
balance = get_balance(account);  
balance -= amount; // 50K-25K
```

```
acquire(lock);
```

```
put_balance(account, balance);  
release(lock);
```

```
balance = get_balance(account);  
balance -= amount; // 25K-25K  
put_balance(account, balance);  
release(lock);  
return balance; // = 0
```

```
return balance; // = 25K
```

שני תהליכים מבצעים את פעולת המשיכה

מה קורה כשהאדום מבקש את המנעול?

מדוע ה return מחוץ לקטע הקריטי?

האם זה נכון?

מימוש מנעולים

- אם היו לנו מנעולים, היה נפלא...
- אבל מימוש מנעול מכיל קטע קריטי:
 - קרא מנעול
 - אם מנעול פנוי, אז
 - כתוב שהמנעול תפוס.

דרכים לממש אנצופים

□ פתרונות תוכנה:

- אלגוריתמים.
- מבוססים על לולאות המתנה (busy wait).

□ שימוש במנגנוני חומרה:

- פקודות מיוחדות שמבטיחות מניעה הדדית.
- לא תמיד מבטיחות התקדמות.
- לא מובטחת הוגנות.

□ תמיכה ממערכת ההפעלה:

- מבני נתונים ופעולות שמהם ניתן לבנות מנגנונים מסובכים יותר.
- בדרך-כלל, מסתמכים על מנגנוני חומרה.

אלגוריתם קופת-חוליות

□ ידוע כאלגוריתם המאפיה (bakery).
[Lamport, 1978]

□ שימוש במספרים:

■ חוט נכנס לוקח מספר.

■ חוט ממתין שמספרו הקטן ביותר נכנס לקטע הקריטי.

חלוקת מספרים: ניסיון 1

Thread i:

```
initially number[i]=0
```

```
number[i]=max{number[1],...,number[n]}+1;
```

```
for all j≠i do
```

```
    wait until number[j]=0 or (number[j]>number[i])
```

```
critical section
```

```
number[i]=0 // Exit critical section
```

□ חוט i ו-j קוראים את המערך בו זמנית

□ בוחרים את אותו מספר

קיפאון!

חלוקת מספרים: שפירת סימטריה

Thread i :

```
initially number[i]=0
```

```
number[i]=max{number[1],...,number[n]}+1;
```

```
for all  $j \neq i$  do
```

```
    wait until number[j]=0 or
```

```
        ((number[j], j) > (number[i], i)) // לקסיקוגרפי
```

```
critical section
```

```
number[i]=0
```

משתמשים במספר החוט (thread id).

בעיה עם האלגוריתם החתוק

Thread 1:

```
number[1]=0
```

```
read{number[1],...,number[n]}
```

Thread 2:

```
number[2]=0
```

```
number[2]=max{number[1],...,number[n]}+1 // =1
```

```
wait until number[1]=0 or ...
```

critical section

```
number[1]=1
```

```
wait until number[2]=0
```

```
or (number[2],2)>(number[1],1))
```

critical section



אין מניעה הדדית!

חלוקת מספרים: ניסיון 3

מחכה שכל החוטים ב-entry יבחרו מספרים

Thread i:

```
initially number[i]=0;
```

```
choosing[i]=true;
```

```
number[i]=max{number[1], ..., number[n]}+1;
```

```
choosing[i]=false;
```

```
for all j≠i do
```

```
    wait until choosing[j]=false;
```

```
for all j≠i do
```

```
    wait until number[j]=0 or
```

```
        ((number[j], j) > (number[i], i));
```

```
critical section
```

```
number[i]=0 // Exit critical section
```


נכונות האלגוריתם

✓ מבטיח מניעה הדדית.

✓ הוגן (אין הרעבה), כניסה לקטע הקריטי (פחות או יותר) לפי סדר הגעה.

✗ מסורבל

- הרבה פעולות, הרבה משתנים
- פונקציה של מספר החוטים
- מבוסס על לולאות המתנה (busy wait)

דרכים אחרות לתיאום בין חוטים / תהליכים...
עזרה מהחומרה.

תיאום בין תהליכים: שיטות מתקדמות

□ שימוש בחומרה למימוש מנעולים

□ מנגנוני מערכת הפעלה לתיאום:

סמפורים, משתני תנאי

מימוש אנצופים: חסימת פסיקות

```
lock_acquire(L) :  
    disableInterrupts()  
    while L≠FREE do  
        enableInterrupts()  
        disableInterrupts()  
    L = BUSY  
    enableInterrupts()
```

□ חסימת פסיקות מונעת החלפת
חוטים ומבטיחה פעולה
אטומית על המנעול

□ למה מאפשרים פסיקות בתוך
הלולאה?

```
lock_release(L) :  
    L = FREE
```

חסימת פסיקות?

□ בעיות במערכת עם מעבד יחיד:

■ תוכנית מתרסקת כאשר הפסיקות חסומות.

■ פסיקות חשובות הולכות לאיבוד.

■ עיכוב בטיפול בפסיקות I/O גורם להרעת ביצועים.

□ במערכות עם כמה מעבדים, לא די בחסימת פסיקות.

■ חוטים יכולים לרוץ בו-זמנית (על מעבדים שונים)

תמיכת חומרה במנעולים

■ $L = \text{false}$ - מנעול פנוי

■ $L = \text{true}$ - מנעול תפוס

```
lock_acquire(L) :  
    while test&set(L)  
        do nop
```

```
lock_release(L) :  
    L = false
```

`test&set(boolvar)`

■ כתוב `true` ל-`boolvar`

והחזר ערך קודם

המתנה בקהלנית

□ spinlock מנעול שיש בו busy waiting:

■ בדוק האם המנעול תפוס (על-ידי גישה למשתנה).

■ אם המנעול תפוס, בדוק שנית.

■ גם בקופת-חולים...

□ מאוד בזבזני.

■ חוט שמגלה כי המנעול תפוס מבזבז זמן cpu.

■ בזמן הזה החוט שמחזיק במנעול לא יכול להתקדם.

priority inversion: כאשר לחוט הממתין עדיפות גבוהה. ☹️

queue lock: מונע busy wait באמצעות ניהול תור של החוטים הממחכים... דורש יותר תמיכה מהחומרה

תמיכת חומרה מתקדמת

compare&swap(mem, R_1 , R_2)

- אם בכתובת הזיכרון mem ערך זהה לרגיסטר R_1 , כתוב את הערך אשר ברגיסטר R_2 והחזר הצלחה
- אחרת החזר כישלון

נתמך בהרבה ארכיטקטורות IA32, Sun.

load-linked / store conditional

- LL(mem) – קרא את הערך בכתובת הזיכרון mem.
- SC(mem, val) – אם לא היתה כתיבה ל- mem מאז ה- LL(mem) האחרון שלך, כתוב ערך val ל- mem והחזר הצלחה (אחרת כשלון)

נתמך בארכיטקטורות של שנות ה-90

■ HP's Alpha, IBM's PowerPC, MIPS4000

אנחנו תיאור אלהים יותר

□ לנהל **תור** של החוטים הממתינים. ☀

□ נמצא במנגנוני תיאום עיליים:

■ סמפורים

■ משתני תנאי

... מוניטורים

סמפור

Babylon English-Hebrew



semaphore •

(פ) לאותות בדגלים, לסמן בזרועות, לאותות
בסמפור

(ש"ע) איתות-דגלים, סימון בזרועות, סמפור, שיטת
תקשורת המבוססת על שימוש בדגלים לאיתות
(לכל אות תנועת דגל מיוחדת); (בתכנת מחשבים)
דגל המשמש לצמצום גישות למשאבים משותפים
בסביבת ריבוי משימות (מעל משאב הנמצא
בשימוש בפני גישת תכניות אחרות)



שני שדות:

■ ערך שלם

■ תור של חוטים /

תהליכים ממתונים

[Dijkstra, 1968]

פעולות על סמור

Babylon German-English dictionary

• Probe (die)

n. trial, test; trying experience; test period, probation, conditional release from jail during which a criminal is under supervision of a probation officer; rehearsal, practice session for a performance; assay

Babylon Dutch-English

• verhogen

v. heighten, put up, make higher, raise, advance, increase, enhance, send up, lift, exalt, augment

wait(semaphore)

■ מקטין את ערך המונה ב-1

■ ממתינים עד שערכו של הסמפור אינו שלילי

■ נקרא גם $P()$, $proben$

signal(semaphore)

■ מגדיל את ערך המונה ב-1

■ משחרר את אחד הממתינים.

■ נקרא גם $V()$, $verhogen$

יותר מ'די חלב עם סמכורים

קוד זהה לשני החוטים.
סמפור OKToBuyMilk, בהתחלה 1.

```
wait( OKToBuyMilk);  
if ( NoMilk ) {  
    Buy Milk;      // critical section  
}  
signal( OKToBuyMilk);
```

אם ערך הסמפור $< 0 \Leftarrow$ ניתן להכנס לקטע הקריטי.

אם ערך הסמפור $\geq 0 \Leftarrow$ הסמפור תפוס, וערכו (בערך מוחלט) הוא מספר הממתינים

סמפור בינארי וסמפור מונה

□ סמפור בינארי

- ערך התחלתי $= 1$; זה גם הערך המקסימאלי.
- מאפשר גישה בלעדית למשאב (בדומה למנעול).

□ סמפור מונה

- ערך התחלתי $0 < N$.
- שולט על משאב עם N עותקים זהים.
- חוט יכול לעבור $wait()$ בסמפור כל עוד יש עותק פנוי של המשאב.

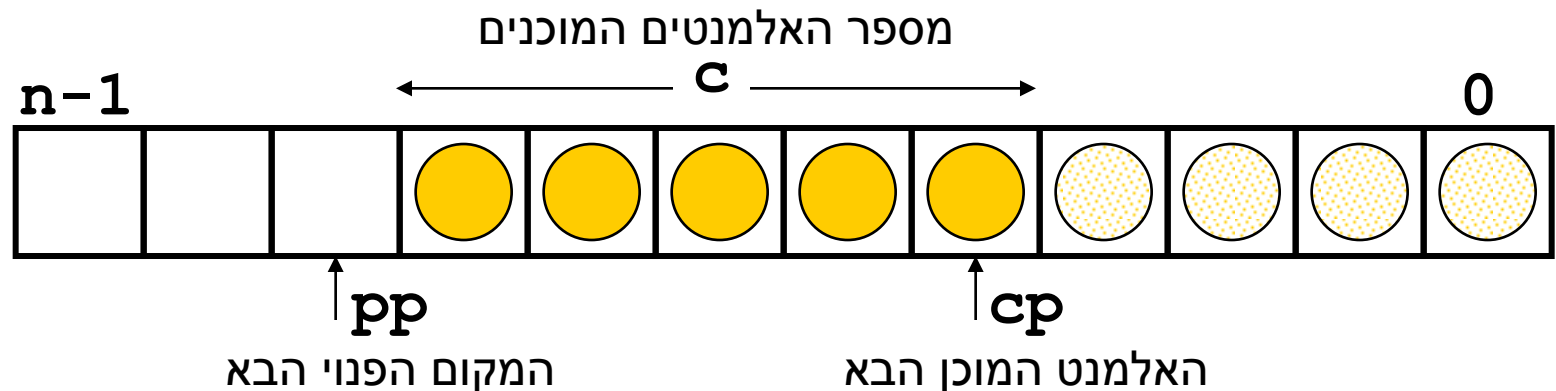
דואמה: קציית יצרון / צרכן

שני חוטים רצים באותו מרחב זיכרון

■ **היצרן** מיצר אלמנטים לטיפול (למשל, משימות)

■ **הצרכן** מטפל באלמנטים (למשל, מבצע את המשימות)

מערך **חסום** (מעגלי) מכיל את העצמים המיוצרים.



פתרון אבציות יצרן / צרכן?

```
global variable  
int c = 0;
```

Producer:

```
repeat  
    wait until (c < n);  
    buff[pp] = new item;  
    pp = (pp+1) mod n;  
    c = c + 1;  
until false;
```

Consumer:

```
repeat  
    wait until (c ≥ 1);  
    consume buff[cp];  
    cp = (cp+1) mod n;  
    c = c - 1;  
until false;
```

ואם ניגשים בו-זמנית ל-c???

יצרן / צרכן עם סמפורים

```
semaphore freeSpace,  
    initially n  
Semaphore availItems,  
    initially 0
```

מספר המקומות הפנויים

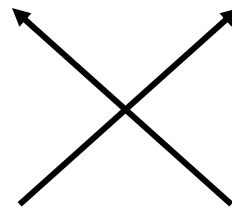
מספר האיברים המוכנים

Producer:

```
repeat  
    wait( freeSpace);  
    buff[pp] = new item;  
    pp = (pp+1) mod n;  
    signal( availItems);  
until false;
```

Consumer:

```
repeat  
    wait( availItems);  
    consume buff[cp];  
    cp = (cp+1) mod n;  
    signal( freeSpace);  
until false;
```



דואלטה: קוראים/כותבים

חוטים קוראים וחוטים כותבים

□ מספר חוטים יכולים לקרוא בו-זמנית.

□ כאשר חוט כותב, אסור שחוטים אחרים יכתבו ו/או יקראו.

טבלת גישה

	Reader	Writer
Reader	✓	✗
Writer	✗	✗

קוראים / כותבים עם סמכורים

```
int r = 0;  
semaphore sRead,  
    initially 1  
semaphore sWrite,  
    initially 1
```

```
Writer:  
[ wait(sWrite)  
  [Write]  
  signal(sWrite)
```

מונה מספר הקוראים
מגן על מונה מספר הקוראים

מניעה הדדית בין קוראים לבין כותבים
(ובין כותבים לעצמם)

```
Reader:  
wait(sRead)  
[ r:=r+1  
  if r=1 then  
    wait(sWrite)  
  signal( sRead)  
  [Read]  
  wait( sRead)  
  [ r:=r-1  
    if r=0 then  
      signal(sWrite)  
    signal( sRead)
```

על (ת'אורט') fe סמולר'ס?

```
struct semaphore_t {  
    int value;  
    queue waitQ;  
}
```

```
void wait(semaphore_t *s){  
    s->value--;  
    if (s->value < 0){  
        enQ(self, &s->waitQ);  
        block }  
}
```

```
void signal(semaphore_t *s){  
    s->value++;  
    if (s->value <= 0){  
        P = deQ( &s->waitQ);  
        wakeup(P) }  
}
```

מ'מוע fe סמכור'ס: ת'קון

```
struct semaphore_t {
    int value;
    queue waitQ;
    lock_t l;
}

void wait(semaphore_t *s){
    lock(&s->l) ;
    s->value--;
    if (s->value < 0){
        enQ(self, &s->waitQ);
        unlock(&s->l) ;
        block }
    else unlock(&s->l) ;
}
```

```
void signal(semaphore_t *s){
    lock(&s->l) ;
    s->value++;
    if (s->value <= 0){
        P = deQ( &s->waitQ) ;
        wakeup(P) }
    unlock(&s->l) ;
}
```

ה'ה busy-waiting ?

□ עדיין יש נעילה: על הגישה לתור השייך לסמפור.

□ עדיין יש busy waiting ...

... אבל הקטע הקריטי קצר:

■ רק לשים / להוריד אלמנט מתור.

■ אפשר לתכנת כך שתהיה עבודה בו-זמנית בשני הקצוות של תור לא-ריק.

⇐ ההשלכות של busy-waiting מוקטנות.

חסרונות של סמכורים

□ לא מפרידים:

■ נעילה.

■ המתנה.

■ ניהול משאבים.

□ רעיון מודרני יותר: נקודות מפגש במשתני תנאי...

פעולות על אשתני תנאי

□ `wait(cond,&lock):`

- שחרר את המנעול (חייב להחזיק בו).
- המתן לפעולת `signal`.
- המתן למנעול (כשחוזר מחזיק במנעול).

□ `signal(cond):`

- הער את **אחד** הממתינים ל `cond`, אשר עובר להמתין למנעול.
- הולך לאיבוד אם אין ממתינים.

□ `broadcast(cond):`

- הער את כל התהליכים הממתינים.
- עוברים להמתין למנעול.
- הולך לאיבוד אם אין ממתינים.

משתני תנאי: הערות

- כאשר תהליך מקבל signal הוא אינו מקבל את המנעול באופן אוטומטי, ועדיין צריך לחכות להשגתו ■
mesa-style
- משתני תנאי הולכים נפלא עם מנעולים.
■ wait(cond,&lock) קודם משחררת את המנעול lock.
- בניגוד לסמפורים, signal לא זוכר היסטוריה.
■ signal(cond) הולך לאיבוד אם אין ממתנים על cond.

דואנא – מ'מלש תור

```
lock QLock ;  
condition notEmpty;
```

□ ממתינים כאשר התור ריק.

```
Enqueue (item):  
lock_acquire( QLock)  
put item on queue  
signal(notEmpty)  
lock_release( QLock)
```

□ נעילה להגן על הגישה
לנתונים.

□ קטע קריטי קצר.

```
Dequeue (item):  
lock_acquire( QLock)  
while queue empty  
    wait(notEmpty, &QLock)  
remove item from queue  
lock_release( QLock)
```

□ משתנה תנאי מאפשר לחכות
עד שיתווסף איבר לתור, מבלי
לבצע busy-wait.

□ למה צריך while?

'תנאי / תנאי שם תנאי תנאי'

```
condition not_full,  
    not_empty;  
lock bLock;
```

producer:

```
lock_acquire(bLock);  
while (buffer is full)  
    wait(not_full, &bLock);  
add item to buffer ;  
signal(not_empty);  
lock_release(bLock);
```

consumer:

```
lock_acquire(bLock);  
while (buffer is empty)  
    wait(not_empty, &bLock);  
get item from buffer  
signal(not_full);  
lock_release(bLock);
```

משתני תנאי + מנצול \cong מוניטור

□ ההקשר המקורי של משתני תנאי [C.A.R. Hoare, 1974]

□ אובייקט (במובן של שפת-תכנות object-oriented),
הכולל פרוצדורת אתחול וגישה.

■ הגישה לאובייקט מקנה שליטה במנעול (באופן לא-מפורש)

■ שליחת signal משחררת את המנעול ומעבירה את השליטה בו
למקבל ה signal.

נתמך בכמה שפות-תכנות מודרניות (ובראשן, Java).

אנחנו תיאום POSIX

□ אובייקטים לתיאום

- נמצאים בזיכרון המשותף.

- נגישים לחוטים של תהליכים שונים.

□ mutex locks (`pthread_mutex_t`)

- יצירה, השמדה: `init`, `destroy`

- `.lock`, `unlock`, `trylock`

□ condition variables (`pthread_cond_t`)

- יצירה, השמדה: `init`, `destroy`

- `.wait`, `signal`, `broadcast`

מנגנוני תיאום Windows NT-ק

□ כל רכיב במערכת ההפעלה הוא אובייקט תיאום

■ ניתן להמתין ו/או לשחרר

□ אובייקטים מיוחדים

■ mutex – מנעול הוגן

■ event – משתנה תנאי (עם אפשרות ל-broadcast כדי להעיר את כל הממתינים)

■ semaphore – סמפור מונה, לא הוגן

■ critical section – light-weight mutex המיועד לחוטים באותו תהליך

טיפול בקיפאון

- בעיית הקיפאון
- הימנעות מקיפאון
- זיהוי קיפאון באמצעות מציאת מעגלים
- אלגוריתם הבנקאי להתחמקות מקיפאון

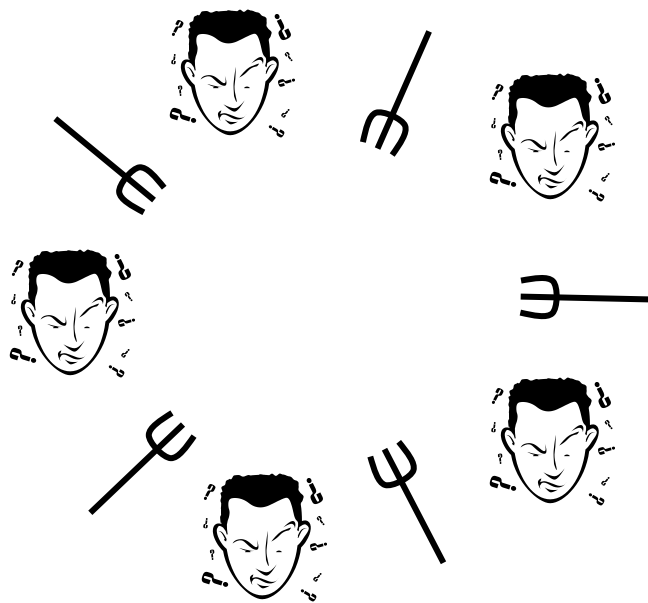
דואל: הפילוסופים הסואצ'ים

□ כל פילוסוף מעביר את חייו ב-

■ חשיבה (thinking)

■ ניסיון להשיג מזלגות (hungry)

■ אוכל (eating)

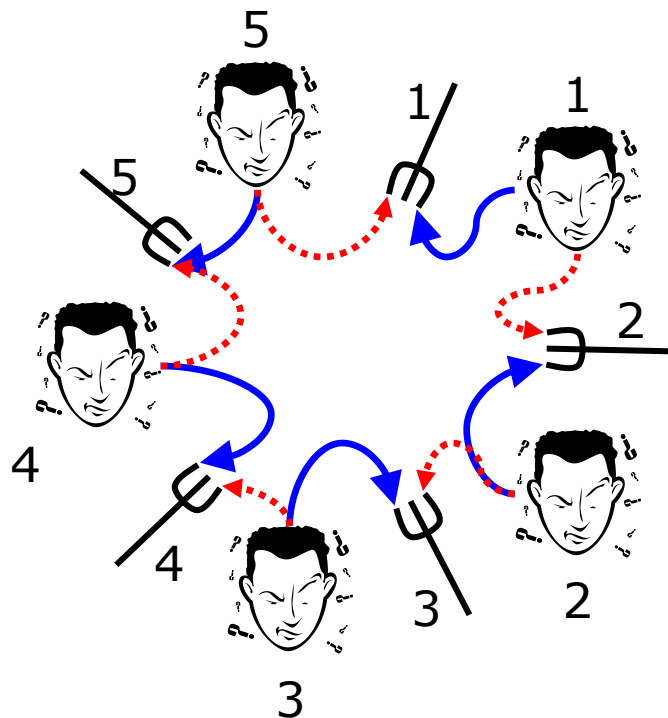


פילוסופים סועדים: עם סמפורים

פיתרון פשוט

ריצה פשוטה?

סמפור $\text{fork}[i]$ לכל מזלג



```
wait(fork[i])  
wait(fork[i+1])
```

eat

```
signal(fork[i])  
signal(fork[i+1])
```

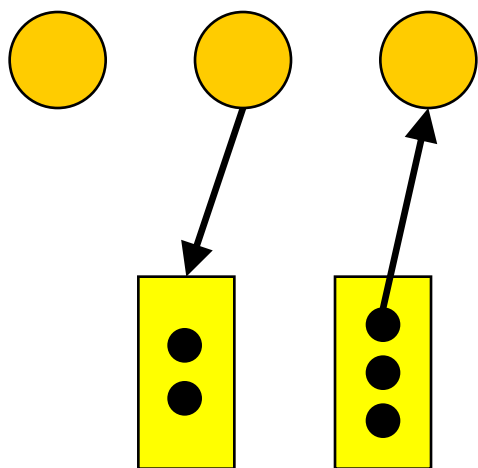
בביצוע מסונכרן כל פילוסוף משתלט כל
המזלג שמימינו ואז ממתין למזלג
שמאלו \Leftarrow קיפאון

דפף בקשות-הקצאות

גרף מכוון (דו-צדדי) המתאר את מצב המערכת בזמן כלשהו

■ עיגול לכל תהליך

■ מלבן עם n נקודות למשאב עם n עותקים



■ קשת מתהליך למשאב

□ התהליך ממתין למשאב

■ קשת מעותק של משאב לתהליך

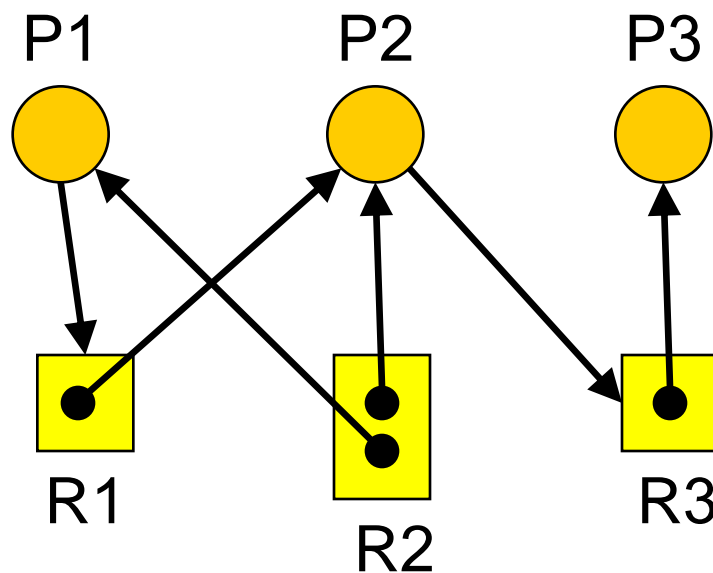
□ עותק של המשאב הוקצה לתהליך

דבר בקשות-הקצאות: AND

□ P1 מחזיק עותק של R2 ומחכה ל-R1

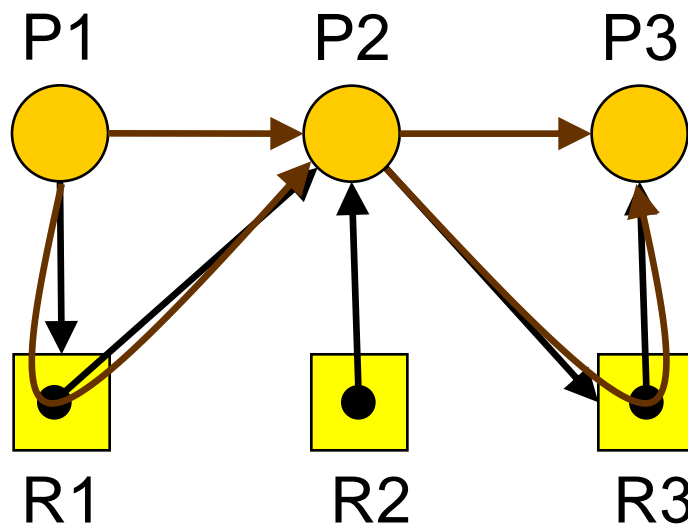
□ P2 מחזיק עותקים של R1 ו-R2 ומחכה ל-R3

□ P3 מחזיק עותק של R3

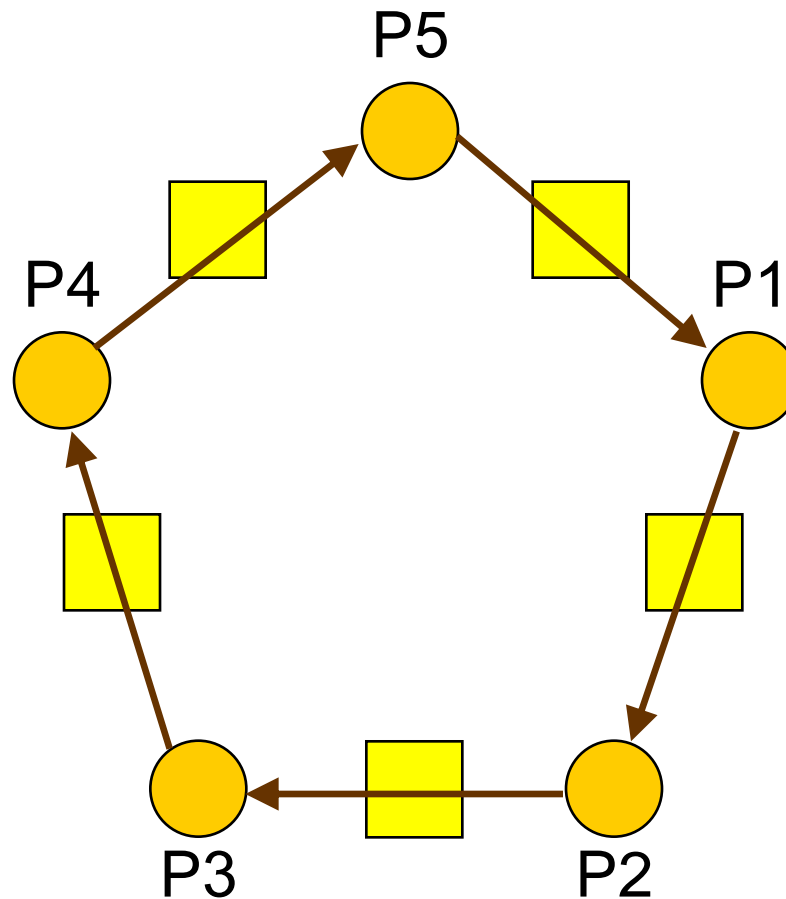


צותק יחיד מכל משאב

ניתן לפשט את גרף בקשות-הקצאות



גורף אבציות הפילוסופים הסוֹאֵרִים



קיפאון

קבוצת תהליכים / חוטים שבה כל אחד ממתין למשאב המוחזק על-ידי מישהו אחר בקבוצה.

מתקיימים התנאים הבאים:

1. יש מניעה הדדית

■ משאבים שרק תהליך אחד יכול לעשות שימוש בהם בו-זמנית

□ כמה עותקים של אותו משאב נחשבים למשאבים נפרדים

■ משאבים שאין צורך במניעה הדדית עבורם אינם יכולים לגרום לקפאון (למשל, read only file)

קיפאון

קבוצת תהליכים / חוטים שבה כל אחד ממתין למשאב המוחזק על-ידי מישהו אחר בקבוצה.

מתקיימים התנאים הבאים:

1. יש מניעה הדדית

2. החזק והמתן

■ תהליך מחזיק משאב ומחכה למשאב אחר שבשימוש אצל תהליך אחר

קיפאון

קבוצת תהליכים / חוטים שבה כל אחד ממתין למשאב המוחזק על-ידי מישהו אחר בקבוצה.

מתקיימים התנאים הבאים:

1. יש מניעה הדדית
2. החזק והמתן
3. לא ניתן להפקיע משאבים

קיפאון

קבוצת תהליכים / חוטים שבה כל אחד ממתין למשאב המוחזק על-ידי מישהו אחר בקבוצה.

מתקיימים התנאים הבאים:

1. יש מניעה הדדית

2. החזק והמתן

3. לא ניתן להפקיע משאבים

4. המתנה מעגלית...

■ תהליכים P_0, P_1, \dots, P_{n-1}

■ P_i מחכה למשאב המוחזק ע"י $P_{(i+1) \bmod n}$

דרכי התמודדות עם הקיפאון

- מניעה (prevention), על-ידי מדיניות שתימנע קיפאון
 - מניעת המתנה מעגלית
 - לחכות לכל המשאבים ביחד
 - פקודת WaitForMultipleObjects ב-Windows NT
 - מקצה את כל המשאבים בבת-אחת, אם כולם פנויים.
- גילוי קפאון (detection)
- היחלצות מקפאון (recovery)
- התחמקות מקפאון (aviodance)
 - שימוש בידע מוקדם על צרכיו של כל תהליך
- רוב מערכות ההפעלה לא דואגות למנוע קיפאון (התעלמות)
 - פעולה כבדה.
 - האחריות נשאת אצל המתכנת .
 - הריגת תהליך מאפשרת להיחלץ מקיפאון.
 - אך עלולה להשאיר את המערכת במצב לא תקין.

מניעת המתנה מצלףית

תהליכים P_0, P_1, \dots, P_{n-1}

■ P_i מחכה למשאב המוחזק ע"י $P_{(i+1) \bmod n}$

■ נקבע סדר מלא בין המשאבים

■ $F(\text{tape})=1, F(\text{printer})=2, F(\text{scanner})=3, \dots$

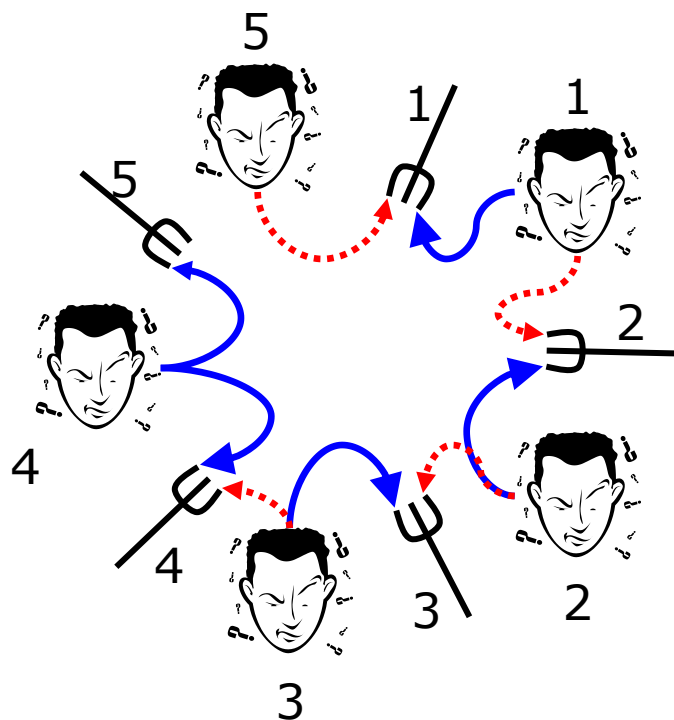
■ תהליכים יבקשו משאבים רק בסדר עולה

■ תהליך שמחזיק את ה-printer לא יוכל לבקש את ה-tape

או... תהליך שמבקש משאב מסדר נמוך חייב לשחרר קודם משאבים מסדר גבוה

חניצת המתנה מצלצית בקציית הפילוסופים הסועדים

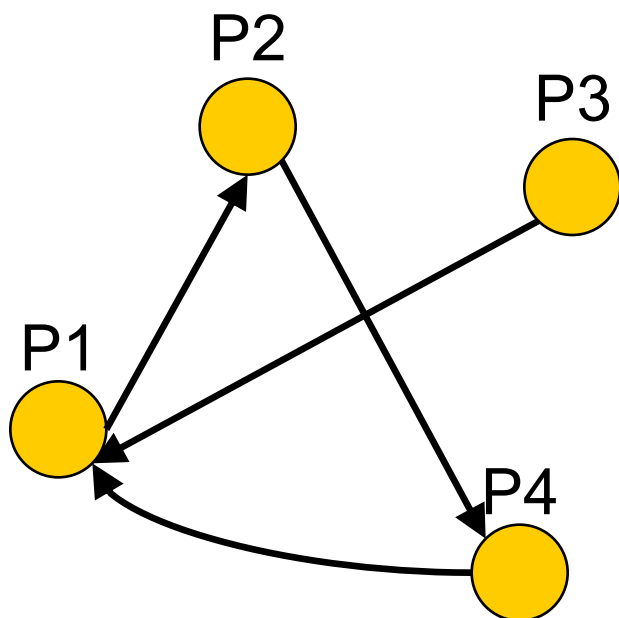
בקשת מזלגות בסדר עולה...



```
if ( i = 5) then
    wait(fork[1])
    wait(fork[5])
else
    wait(fork[i])
    wait(fork[i+1])
eat
signal(fork[i+1])
signal(fork[i])
```

גילוי קיפאון עם צומק יחיד

קיפאון קורה כשיש מעגל מכוון בגרף בקשות-הקצאות



נובע מתנאי "החזק והמתן"
ו-"המתנה מעגלית"

גילוי קיפאון על-ידי זיהוי מעגל

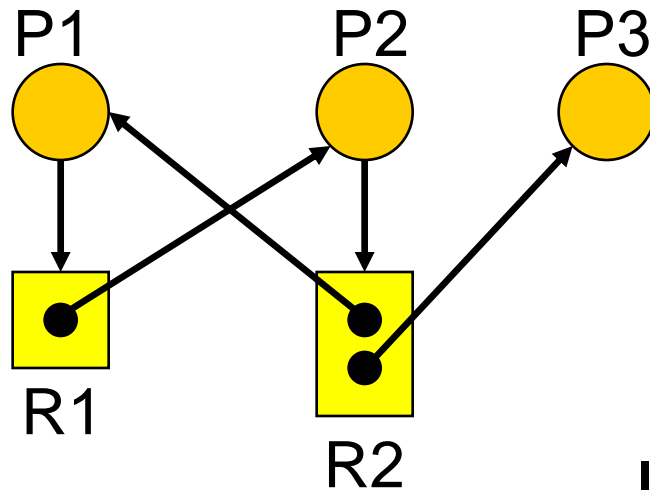
אלגוריתם ידוע למציאת מעגל בגרף מכוון, המבוסס על מיון טופולוגי

סיבוכיות $O(m)$, כאשר m מספר הקשתות

מספר עותקים מכאן

לא חייב להיות קיפאון גם אם יש מעגל מכוון בגרף בקשות-
הקצאות

ז"א, יכול להיות מעגל בלי שיהיה קיפאון.



גם עבור מערכת עם כמה עותקים
מכל משאב קיים אלג' לגילוי קיפאון
אבל לא נלמד אותו

גילוי קיפאון כאשר המשאבים מחולקים למחלקות

א. סימונים

ישנם n תהליכים, והם מסומנים במספרים P_1 עד P_n .
ישנן m מחלקות משאבים המסומנות במספרים R_1 עד R_m .

ב. מבני נתונים

1. E הוא מערך חד-ממדי בגודל m . בכל תא E_j מאוחסן מספר המשאבים הכולל השייך למחלקת המשאבים j ($1 \leq j \leq m$).
 2. A הוא מערך חד-ממדי בגודל m . בכל תא A_j מאוחסן מספר המשאבים החופשיים (שלא הוקצו עדיין) מהמחלקה j .
 3. C הוא מערך דו-ממדי בגודל $m * n$. בתא C_{ij} מאוחסן מספר המשאבים מהמחלקה j שכבר הוקצו לתהליך i .
 4. R הוא מערך דו-ממדי בגודל $m * n$. בתא R_{ij} מאוחסן מספר המשאבים מהמחלקה j הנדרשים עדיין על ידי התהליך i .
- ג. מכיוון שכל משאב יכול להיות חופשי או מוקצה, הרי שהסכום של מספר המשאבים מהמחלקה j שכבר הוקצו לאחד התהליכים ומספר המשאבים ממחלקה זו שעדיין חופשיים הוא תמיד E_j .
- ד. אנו מגדירים יחס **בין שני וקטורים** A ו B שווים באורכם על ידי $A \leq B$ כאשר לכל ומתקיים $A_i \leq B_i$.

אלגוריתם לגילוי מצב הקיפאון

```
DetectDeadlock(){ {P = {P1, P2, ..., Pn};  
  while (P is not empty){  
    runnable_process_exists = false;  
    for each p in P {if (Rp <= A){A = A + Cp; P = P \ {Pp};  
      runnable_process_exists = true; } }// at least one process exists  
    if (runnable_process_exists == false) report processes in P as deadlocked;  
    }report no deadlock;}
```

1. מחפשים תהליך p שנמצא במצב של המתנה למשאבים ואשר הווקטור R_p המייצג את בקשותיו (ווקטור השורה p במערך R) קטן או שווה לווקטור A המייצג את המשאבים הזמינים ($R_p \leq A$). במילים אחרות, אנו מחפשים תהליך הממתין למשאבים, שניתן להיענות לכל בקשותיו. תהליך כזה יכול להמשיך ולסיים את פעולתו ללא חשש מקיפאון.
2. אם מצאנו לפחות תהליך אחד כזה, אנו מחברים חיבור וקטורי ומציבים : $A = A + C_p$. כמו כן אנו מעדכנים את הקבוצה P כדי שלא תכיל את התהליך P_p ניתן לתאר במילים את מה שנעשה בשלבים אלה: התהליך P_p יכול להמשיך לרוץ כי הבקשות שלו מסופקות, ובסיום הוא יחזיר את כל המשאבים שהחזיק בהם לשימושם של תהליכים אחרים.
3. כאשר לא ניתן עוד למצוא תהליך P_p שניתן לספק את מבוקשו, ויחד עם זאת הקבוצה P איננה ריקה, הדבר מצביע על קיום מצב הקיפאון. התהליכים שנותרו בקבוצה P נמצאים במצב קיפאון.
4. כאשר האלגוריתם מסתיים והקבוצה P ריקה, ברור שאין מצב קיפאון.

היחלוצות מקיפאון Recovery

- ביטול כל התהליכים המצויים במצב קיפאון.
- ביטול תהליכים אחד-אחד עד להיחלוצות.
- הפקעת משאבים.

מציאת קבוצה אופטימאלית (של תהליכים או משאבים) לביטול היא בעיה קשה מאוד.

Avoidance **התחמקות מקיפאון**

□ דואגים להשאיר את המערכת במצב בטוח

■ תהליכים מצהירים על כוונתם לבקש משאבים

■ אסור לתהליך לבקש משאב שלא הצהיר עליו

מצב הוא **בטוח** אם קיימת סדרת הקצאות לכל המשאבים שהוצהרו, מבלי להיכנס לקיפאון.

□ אם היענות לבקשה עלולה להכניס את המערכת למצב לא בטוח, אז הבקשה נידחת.

□ אלגוריתם שמרני

■ הנחות מחמירות לגבי דרישות התהליכים

התחלקות מקיפאון: דולנד

- P_1 ו- P_2 מצהירים על כוונה לבקש את המדפסת והדיסק
- P_1 מבקש את המדפסת
 - הבקשה נענית
- P_2 מבקש את הדיסק
 - הבקשה נדחית...
- אם בהמשך P_2 יבקש את המדפסת ו- P_1 את הדיסק, אזי יהיה קיפאון
- בבעית הפילוסופים הסועדים - שמים את המזלגות במרכז השולחן
 - לוקח מזלג, אלא אם כן:
 - זה המזלג האחרון ולאחד הממתינים עדיין חסר מזלג אחד
- ובאופן כללי, אלגוריתם הבנקאי...

התחמקות: אפלטון ריתם הנקאי

□ כאשר תהליך מגיע למערכת, מצהיר כמה עותקים ירצה (לכל היותר) מכל משאב.

□ אם תהליך מבקש עותק ממשאב, הבקשה תיענה רק אם

■ יש מספיק עותקים פנויים.

■ לא חורג מהצהרה ההתחלתית.

■ ההקצאה משאירה את המערכת במצב **בטוח**: תוכל לספק את הבקשות של תהליכים שכבר נמצאים במערכת עד למקסימום המוצהר לכל סוג משאב

אלגוריתם הבנקאי: משתנים

m מספר המשאבים, n מספר התהליכים

Available: vector of size m
the number of available copies of each resource type

Max: $n \times m$ matrix
the maximum demand
of each process for each resource type

Allocation: $n \times m$ matrix
number of currently allocated copies
of each resource type to each process

Need: $n \times m$ matrix
the remaining copies that might be needed by each process
Need = Max - Allocation

Work and Finish: (temporary) vectors of size m and n

אלגוריתם הבנקאי: פסאודו-קוד

לפני אישור בקשה למשאב:
ודא שהמערכת נשארת במצב בטוח אם הבקשה מתקבלת

Initialize

Work := Available;

Finish := [false, ..., false];

While (Finish[i] = false

& Need[i] ≤ Work), for some i

Work := Work + Allocation[i];

Finish[i] := true;

End while

The system is safe if and only if

Finish[i] = true, for all i

ניתן יהיה להיענות
לדרישותיו בעתיד

שחרר את המשאבים
שמחזיק כעת

אלגוריתם הבנקאי – דוגמה

□ האם המצב הבא של המערכת הוא "בטוח"?

	Allocation			Max			Available			Need			Finish
	A	B	C	A	B	C	A	B	C	A	B	C	
P1	0	1	0	3	1	1				3	0	1	F
P2	2	1	0	2	1	4				0	0	4	F
P3	1	0	3	3	2	3				2	2	0	F
Total	3	2	3				0	3	4				

Work	0	3	4
------	---	---	---

□ ניתן לצמצם את P2

אלגוריתם הבנקאי – דוגמה

□ האם המצב הבא של המערכת הוא "בטוח"?

	Allocation			Max			Available			Need			Finish
	A	B	C	A	B	C	A	B	C	A	B	C	
P1	0	1	0	3	1	1				3	0	1	F
P2	2	1	0	2	1	4				0	0	4	T
P3	1	0	3	3	2	3				2	2	0	F
Total	3	2	3				0	3	4				

□ ניתן לצמצם את P2

□ עכשיו, ניתן לצמצם את P3

Work	2	4	4
------	---	---	---

אלגוריתם הבנקאי – דוגמה

□ האם המצב הבא של המערכת הוא "בטוח"?

	Allocation			Max			Available			Need			Finish
	A	B	C	A	B	C	A	B	C	A	B	C	
P1	0	1	0	3	1	1				3	0	1	F
P2	2	1	0	2	1	4				0	0	4	T
P3	1	0	3	3	2	3				2	2	0	T
Total	3	2	3				0	3	4				

□ ניתן לצמצם את P2

□ עכשיו, ניתן לצמצם את P3

□ עכשיו, ניתן לצמצם את P1

Work	3	4	7
------	---	---	---

אלגוריתם הבנקאי – דוגמה

□ האם המצב הבא של המערכת הוא "בטוח"?

	Allocation			Max			Available			Need			Finish
	A	B	C	A	B	C	A	B	C	A	B	C	
P1	0	1	0	3	1	1				3	0	1	T
P2	2	1	0	2	1	4				0	0	4	T
P3	1	0	3	3	2	3				2	2	0	T
Total	3	2	3				0	3	4				

□ ניתן לצמצם את P2

□ עכשיו, ניתן לצמצם את P3

□ עכשיו, ניתן לצמצם את P1

Work

3	5	7
---	---	---



Watchdog

□ מכניזם פרקטי, הממומש בחומרה

■ מונה חומרה מיוחד שסופר אחורה

■ כשהמונה מתאפס, מבצע reset להתקן (מכבה ומדליק ההתקן מחדש)

■ החוטים מדי פעם ומעדכנים את המונה לערך גבוה התחלתי

■ אם יש קפאון – החוטים לא יעדכנו את המונה ויהיה reset

□ מחייב שיתוף פעולה של החוטים העובדים

□ אם רוצים לוודא שאף חוט לא מעורב בשום קפאון, צריך להשתמש ברגיסטר נפרד לכל חוט

מימון Watchdog במכונה

שיטה 1: כמו בחומרה, רק שחוט מיוחד בודק את המונה ואם מגלה שלא עודכן הרבה זמן, הורג את התהליך

- מחייב שיתוף פעולה של כל החוטים העובדים

שיטה 2: חוט מיוחד A מנסה, אחת לכמה זמן, לתפוס את כל המשאבים במערכת לפי סדר נתון ואז לשחרר אחת לכמה זמן כותב את הזמן הנוכחי למונה

חוט אחר B בודק את המונה, ואם מגלה שלא עודכן הרבה זמן, מניח שחוט A בקיפאון, הורג את התהליך (וכל החוטים)

- מחייב ידע של חוט A לגבי כל משאבי התוכנית, אבל לא מחייב שיתוף פעולה של כל החוטים העובדים
- השיטה פוגעת קצת בביצועי התוכנית (intrusive) כי A תופס משאבים

פסיקות

- סוגי פסיקות
- איך מערכת ההפעלה מטפלת בפסיקות
- דוגמא: קלט בעזרת פסיקות

מהן פסיקות?

□ פסיקות (interrupts) הן אירועים הגורמים למעבד להשעות את פעילותו הרגילה ולבצע פעילות מיוחדת.

□ שימושים:

- מימוש קריאות מערכת-הפעלה.
- קבלת מידע וטיפול ברכיבי חומרה (שעון, חומרת קלט/פלט...).
- טיפול בתקלות (חלוקה באפס, גישה לא חוקית לזיכרון...).
- אכיפת חלוקת זמן מעבד בין תהליכים

סוגי פסיקות

□ פסיקות **אסינכרוניות** נוצרות על-ידי רכיבי חומרה שונים:

- ללא תלות בפעילותו הנוכחית של המעבד.
- למשל: פעימה של שעון המערכת, לחיצה על מקש במקלדת...

□ פסיקות **סינכרוניות** נוצרות בשל פעילות של המעבד:

- למשל, כתוצאה מתקלות שונות.
- נקראות גם **חריגות** (exceptions).

□ סוג אחר של פסיקות סינכרוניות הוא **פסיקות יזומות**, אשר נקראות

גם **פסיקות תוכנה** (software interrupts):

- נוצרות על-ידי הוראות מעבד מיוחדות (למשל int).
- שימושים: מימוש קריאות מערכת, debugging.

הלצאות שנייה

- ביצוע הוראות מסוימות יכול להכשל ולגרום לפסיקה סינכרונית
- מעבדים מודרניים מאפשרים למערכת ההפעלה, על-ידי טיפול בפסיקה, לתת להוראות "הזדמנות שנייה" להתבצע (נקראות restartable instructions)
 - ההוראה נכשלה ⇐ פסיקה
 - מערכת-ההפעלה מטפלת בפסיקה
 - מריצים שוב אותה ההוראה (שהפעם אמורה להצליח).
- שימושי למימוש זיכרון וירטואלי
 - בהמשך הקורס

את המעבד מאלף שהיו פסיקות לאחר ביצוע פאולה...

□ שומר על המחסנית את כתובת החזרה.

- בד"כ כתובת ההוראה הבאה.
- במקרה של "הזדמנות שנייה", כתובת ההוראה שגרמה לפסיקה.
- ב Linux, הערכים נשמרים במחסנית הגרעין של התהליך, ולא במחסנית המשתמש.

□ אולי מידע נוסף.

- רגיסטרים מיוחדים
- מידע נוסף לגבי הפסיקה, למשל, סוג השגיאה המתמטית.

□ מפעיל את **שגרת הטיפול** של הפסיקה

- ב-Linux ניתן להפעיל מספר שגרות בעקבות פסיקה
- השגרה מוגדרת עבור (התקן, פסיקה) ולא עבור (פסיקה) בלבד
- מספר התקנים יכולים לחלוק את אותה פסיקה.

איך מפעילים את שורת הטיפול בפסיקה?

- לכל פסיקה שגרת טיפול משלה, הפועלת בהתאם לסוג הפסיקה
 - למשל, קוראת את התו שנלחץ במקלדת ושומרת אותו במקום מתאים בזיכרון
- לכל פסיקה יש מספר שונה (נקרא לפעמים interrupt vector).
- למעבד יש גישה לטבלה של מצביעים לשגרות טיפול בפסיקה.
 - מספר הכניסות בטבלה כמספר וקטורי הפסיקות.
 - במעבדי IA16, הטבלה שמורה בכתובת 0000:0000 (ממש בתחילת הזיכרון) מכילה 256 כניסות של 4 בתים כ"א (4 בתים = גודל מצביע), סה"כ 1KB.
 - טבלת הפסיקות מאותחלת על-ידי חומרת המחשב (ה-BIOS).
 - בטעינה, מערכת-ההפעלה מעדכנת את הטבלה, ומחליפה חלק מהשגרות.
 - מערכות-הפעלה מודרניות לא מסתמכות כלל על שגרות הטיפול של ה-BIOS, ומחליפות את כל הטבלה.

מי מריץ את שצרת הטיפול בפסיקה?

- הפסיקה קורה בזמן שתהליך כלשהו רץ.
 - הפסיקה אינה בהכרח קשורה לתהליך זה.
 - למשל, נגרמה על-ידי חומרה שתהליך אחר משתמש בה.
 - קוד הטיפול בפסיקה שייך ל-kernel ולא לתהליך.
- ועדיין, קוד הטיפול מורץ בהקשר של התהליך הנוכחי...
- לא מתבצעת החלפת תהליכים כדי לטפל בפסיקה!
 - בדרך-כלל, קוד הטיפול בפסיקה לא מתייחס כלל לתהליך הנוכחי, אלא למבני נתונים גלובליים של המערכת.

הציות הטיפול בפסיקות



□ פסיקות א-סינכרוניות יכולות להגיע בכל זמן שהוא.

■ המשתמש יכול ללחוץ על המקלדת בכל רגע...

■ מצד אחד, חשוב לטפל בהן במהירות האפשרית...

□ בעיקר כדי לא לעכב התקני קלט-פלט.

□ זמן הטיפול עשוי להיות רב (למשל, מידע שהגיע מהדיסק).

■ מצד שני, טיפול שדורש זמן רב ומתחיל מיד, מתבצע על חשבון התהליך הנוכחי

□ כמו כן, יש לצמצם את זמן חסימת הפסיקות למינימום

□ בנוסף, יש לתמוך בקיבוע: פסיקה שמגיעה בזמן הטיפול בפסיקה אחרת

פתרון (חלקי): טיפול דו-שלבי בפסיקות אסינכרוניות

□ חלק עליון: (top half)

- מורץ כשגרת הטיפול בפסיקה, באופן מהיר ככל האפשר.
- רושם (בתור מיוחד) את העובדה שיש לתת לחומרה הרלוונטית טיפול הולם (למשל, לקרוא את כל המידע שהגיע מהדיסק).

□ חלק תחתון: (bottom half)

- מורץ בזמן מאוחר יותר, ומבצע את "העבודה השחורה" של הטיפול בפסיקה.
- פסיקות אחרות יכולות להתרחש תוך-כדי פעולתו.

□ המערכת שומרת תור של שגרות להרצה (חלקים תחתונים).

- שגרת הטיפול בפסיקה פשוט מוסיפה לתור זה את החלק התחתון המתאים.
- בנקודות זמן מסוימות (נניח, בכל החלפת הקשר) המערכת בודקת אם התור אינו ריק, ומריצה את כל השגרות הממתינות.

פרשור פסיקות: שינוי שגרת הטיפול

- ניתן לשנות את שגרת הטיפול בפסיקה בזמן פעולת המערכת.
 - החלפת השגרה נעשית פשוט על-ידי שינוי הטבלה.
- אבל מה אם רוצים להוסיף לשגרה הנוכחית, מבלי לבטל אותה?
 - למשל, שינוי שגרת הטיפול במקלדת להשמיע קליק לאחר כל לחיצת מקש.
 - בהחלפת השגרה הקיימת, יושמעו קליקים, אבל הלחיצות עצמן לא תטופלנה!
- לפני שמעדכנים את טבלת הפסיקות, שומרים את כתובת השגרה הנוכחית.
- שגרת הטיפול החדשה קוראת לשגרה הישנה (בכתובת שנשמרה):
 1. קודם קוראים לשגרה המקורית, ורק אחר-כך מבצעים את השגרה החדשה.
 2. **או** קודם השגרה החדשה, ורק אחר-כך מבצעים את השגרה המקורית.
 - מאפשר לא לבצע את הפעולה המקורית במקרים מסוימים.

טיפול בפסיקות

כמערכת מרוכבת-מאגדים

□ פסיקות סינכרוניות: כל מעבד מטפל בפסיקות שיצר הקוד שהוא מריץ.

□ פסיקות אסינכרוניות: איזה מעבד יתעכב כדי לטפל בקלט-פלט?

■ **חלוקה סטטית:** לכל מעבד קבוצה של (סוגי) פסיקות שהוא אחראי עליה.

■ **חלוקה דינאמית:** המעבד שמריץ את התהליך עם העדיפות הגרועה יותר, יטפל בפסיקה

□ round-robin במקרה של שוויון

□ בנוסף, במערכות מרובות-מעבדים יש גם פסיקות המשמשות

להעברת מידע בין המעבדים עצמם

■ **Inter-Processor Interrupts**

■ מעבד רואה IPIs של מעבדים אחרים כפסיקות א-סינכרוניות רגילות.

דולא: קריאה מהדיסק 1

□ המשתמש מבצע fread כדי לקרוא מידע מהקובץ



fread(...)

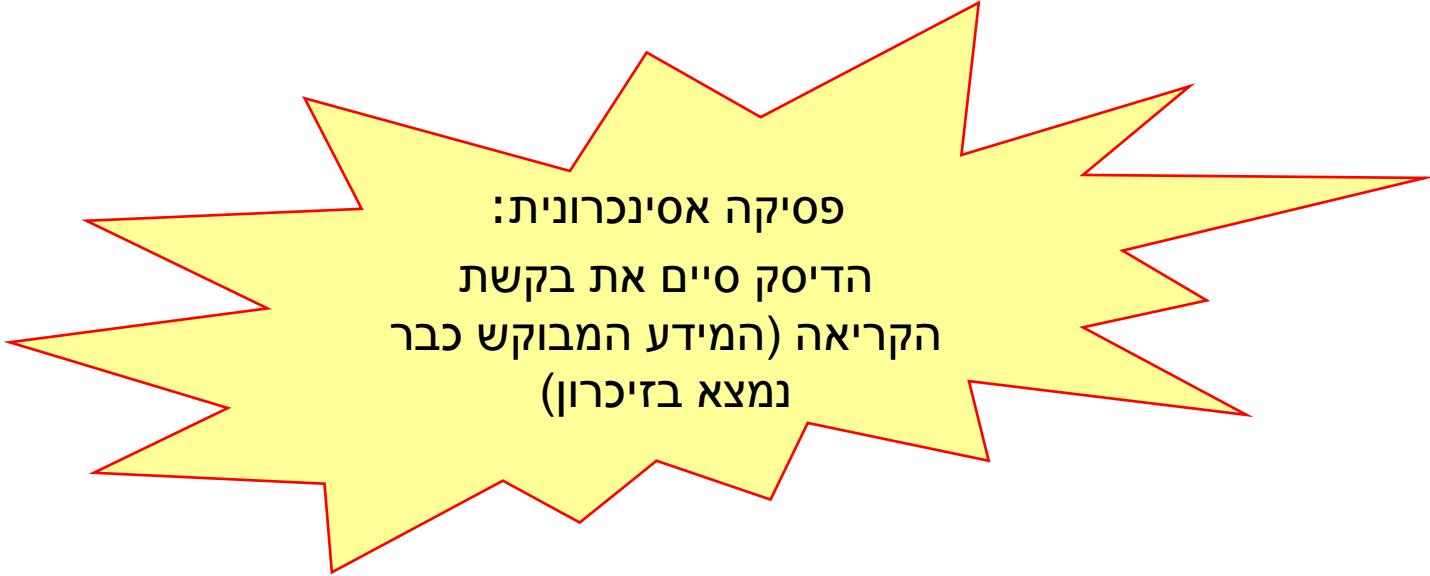
מציאת המקום על הדיסק בו שמור המידע
(ע"י קריאת טבלאות, למשל)

מעבירים לדיסק בקשה:
נא לקרוא את המידע מכתובת כך-וכך, ולשמור
במיקום זה-וזה בזיכרון...

דולא: קריאה מהדיסק 2

□ התהליך שביצע את הקריאה ממתין

□ בינתיים, רצים תהליכים אחרים...



פסיקה אסינכרונית:
הדיסק סיים את בקשת
הקריאה (המידע המבוקש כבר
נמצא בזיכרון)

דואל: קריאה מהדיסק 3

- מופעלת השגרה לטיפול בפסיקות הדיסק.
- מוסיפה את החצי התחתון הרלוונטי לרשימה, ובזאת מסתיים הטיפול בפסיקה.
- בהמשך, מורץ החצי התחתון:
 - בודק שהמידע שהדיסק העביר לא כולל דיווח על שגיאות.
 - מעביר את התהליך שביצע את הקריאה ממצב המתנה אל תור המוכנים.
 - אולי גם מעדכן את העדיפות של אותו תהליך.
- התהליך חוזר מהקריאה fread ומקבל את המידע.

ניהול זיכרון

□ מבוא: מטרות ניהול הזיכרון.

□ מנגנונים:

- מרחב כתובות וירטואלי / פיזי.
- חלוקת זכרון קבועה מול דפדוף.
- ניהול טבלת הדפים.
- מדיניות החלפת דפים.

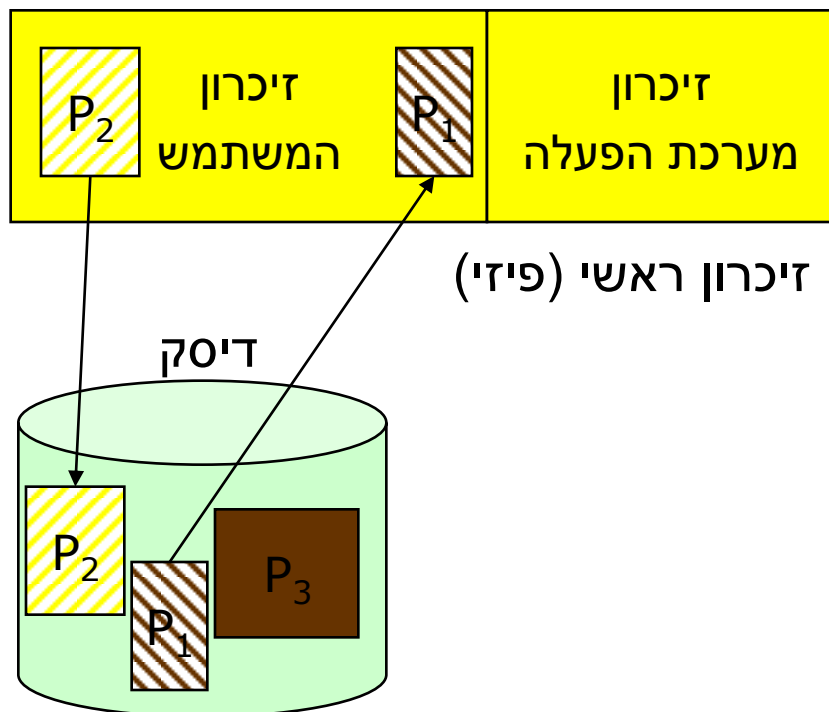
ניהול הזיכרון

□ מערכת ההפעלה צריכה לנהל את השימוש בזיכרון:

- חלק מהזיכרון מוקצה למערכת ההפעלה עצמה.
- שאר הזיכרון מתחלק בין התהליכים הרצים כרגע.
- כאשר תהליך מתחיל צריך להקצות לו זיכרון.
- כאשר תהליך מסיים, ניתן לקחת בחזרה זיכרון זה.

□ מערכת ההפעלה צריכה למנוע מתהליך גישה לזיכרון של תהליכים אחרים.

Swapping

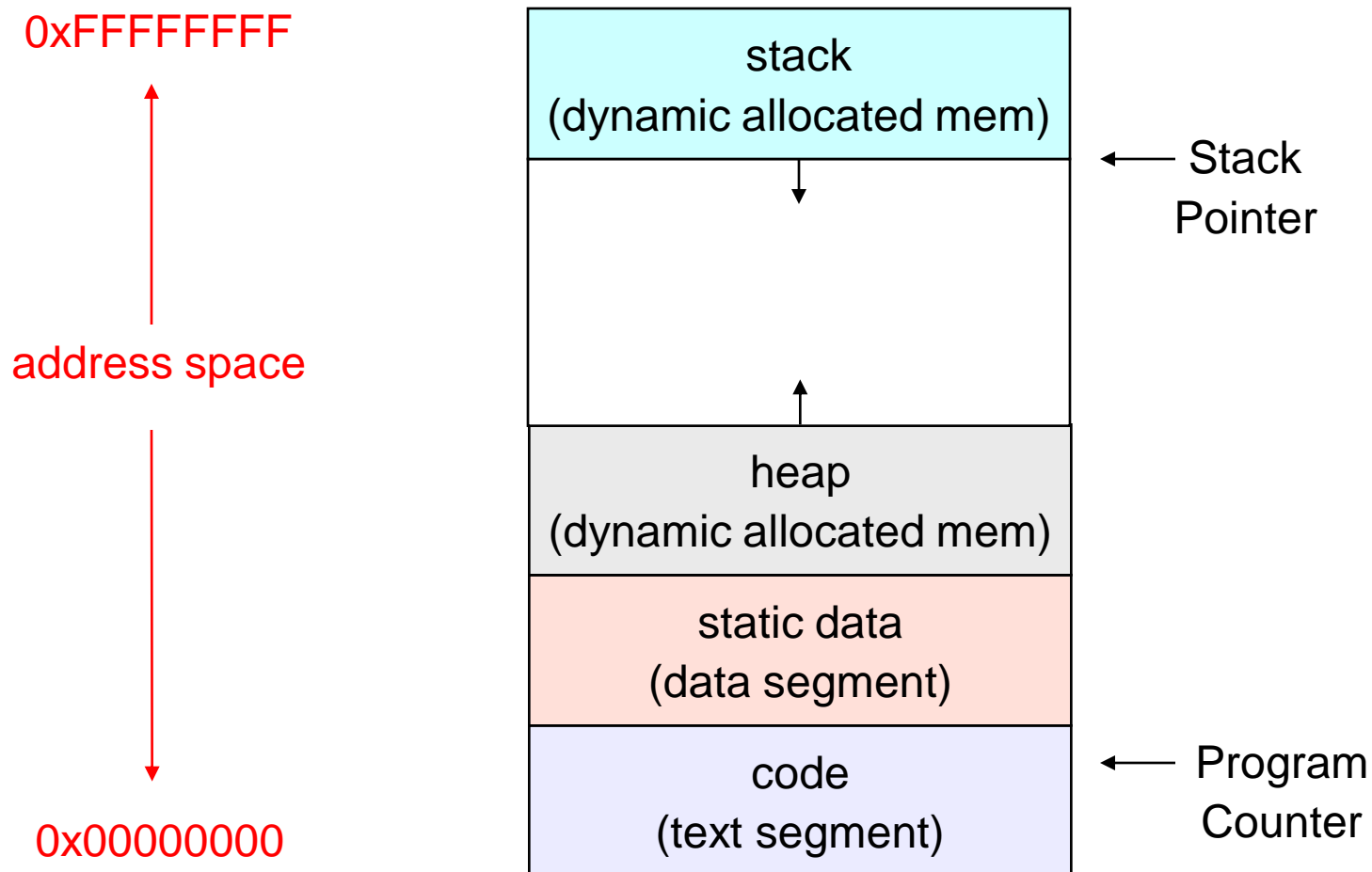


□ זיכרון של תהליך שאינו רץ מועבר לדיסק (swap-out).

□ כאשר תהליך חוזר לרוץ, מקצים לו מחדש מקום ומביאים את הזיכרון שלו (swap-in).

דורש מנגנון תמיכה...

מרחב הכתובות של התהליך (תזכורת)



מרחב הכתובות ef התהליך

□ בעיקרון, כל מרחב הכתובות צריך להיות זמין בזיכרון הפיזי של המחשב, כאשר התהליך רץ...
כתובת של 32 ביטים ← מחשב עם זיכרון פיזי בגודל $2^{32} = 4GB$?
... ומה עם דרישות הזיכרון של תהליכים אחרים?

□ האם תהליך באמת צריך את כל הזיכרון הזה?
... בכלל משתמש בו?

□ השטח הכולל שבשימוש קטן בהשוואה למרחב הזיכרון כולו

← מקצים זיכרון רק אם משתמשים בו.

חיסכון בליכרון

□ האם תהליך באמת צריך את כל הזיכרון הזה כל הזמן?
... ואם השתמש פעם אחת, האם ייגש אליו שנית?

□ קוד איתחול.

□ מחסנית שגדלה מאוד, ואז קטנה.

□ זיכרון דינאמי משוחרר על-ידי התהליך

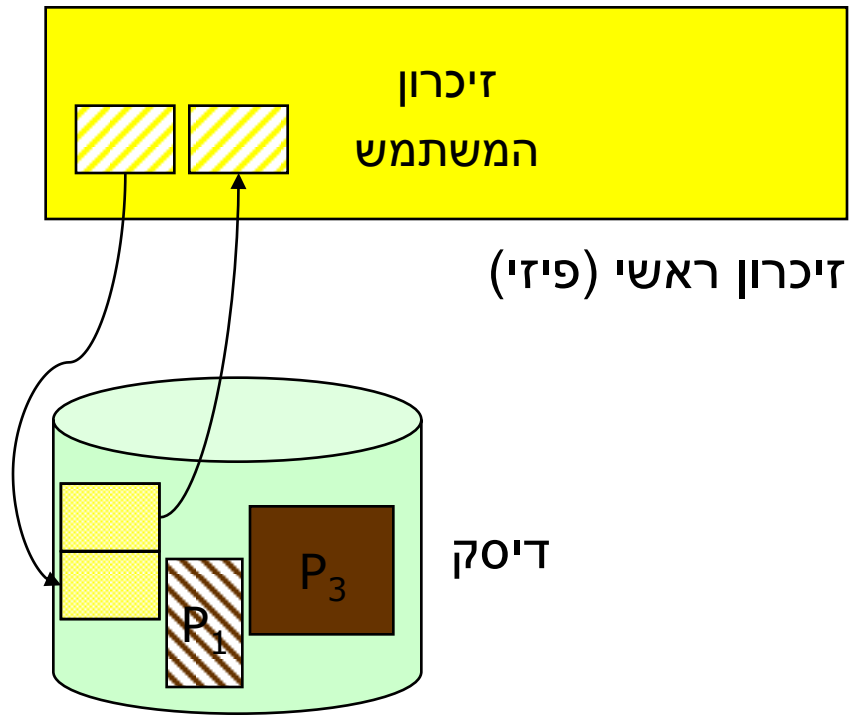
□ משתנים שמשתמשים בהם רק בשלב מסוים של הקוד

□ עקרון הלוקליות: תהליך ניגש רק לחלק מזערי של הזיכרון
שברשותו בכל פרק זמן נתון

□ צריך לאחסן ערכים של משתנים, גם אם לא משתמשים
בהם...

אבל לא בזיכרון הפיזי!

לרוק אותו לדיסק!



הדיסק מכיל חלקים מהזיכרון של תהליך שרץ כרגע.

■ צריך לזכור מה נמצא בזיכרון הפיזי ואיפה (ונמצא בדיסק).

■ צריך לבחור מה להעביר לדיסק.

■ צריך לזכור איפה שמנו חלקי זיכרון בדיסק, כדי לקרוא אותם בחזרה, אם נצטרך אחר-כך.

זיכרון וירטואלי (ממחשבי הזיכרון)

□ מרחב כתובות מלא לכל תהליך.

■ יכול להיות גדול מגודל הזיכרון הפיזי.

■ רק חלקי הזיכרון הנחוצים כרגע לתהליך נמצאים בזיכרון הפיזי.

□ תהליך יכול לגשת רק למרחב הכתובות שלו.

□ מרחב כתובות **פיזי**: מוגבל בגודל הזיכרון הפיזי במחשב.

□ מרחב כתובות **וירטואלי** אותו רואה כל תהליך.

■ אינו מוגבל בגודלו (אלא על-ידי גודל הדיסק).

הפרדה בין תהליכים

הגנה.

- לא מאפשרים לתהליך גישה לנתונים של תהליך אחר.
- כתובות מתורגמות לתוך מרחב הכתובות הוירטואלי של תהליך זה בלבד.

שמירה על אי-תלות בביצועים.

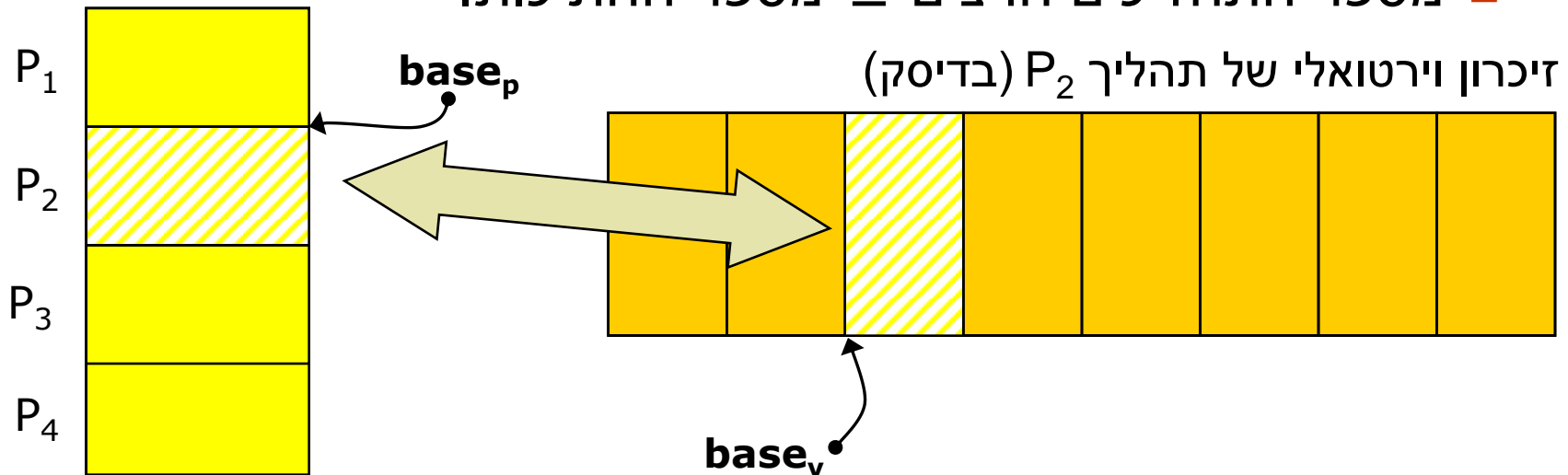
- מערכת ההפעלה מחלקת משאבים מצומצמים בין כמה תהליכים.
- דרישות הזיכרון (פיזי) של תהליך לא על-חשבון תהליך אחר.

שיטה ישנה: חלוקה קבועה

■ מערכת ההפעלה מחלקת את הזיכרון הפיזי לחתיכות בגודל קבוע $size$ שיכולות להכיל חלקים ממרחב הכתובות של תהליך.

■ לכל תהליך מוקצית חתימת זיכרון פיזי.

זיכרון ראשי (פיזי)



הציות עם חלוקה קבועה

□ החלפה של חתיכת זיכרון שלמה בבת אחת.

□ עבודה עם כתובות זיכרון רציפות:

■ אם גדול מספיק להכיל כל מה שצריך (וגם מה שבאמצע)

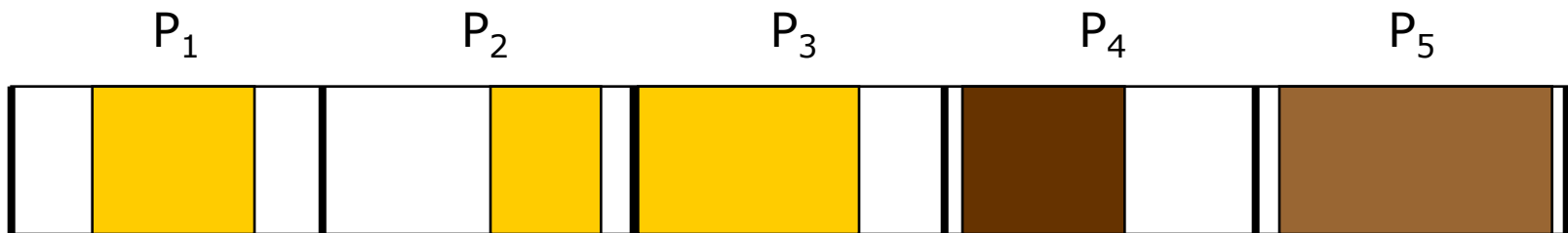
← כל החלפה לוקחת הרבה זמן.

■ מאפשר מעט חתיכות שונות

← מעט מידי תהליכים רצים בו-זמנית.

שבר פנימי

דרישות זיכרון שונות לתהליכים שונים.



חלק מחתיכת הזיכרון של התהליך מבוזבז
(internal fragmentation)

שיטה ישנה נוספת: חלוקה משתנה

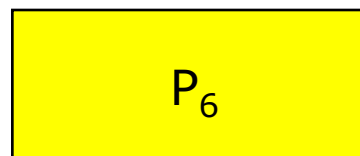
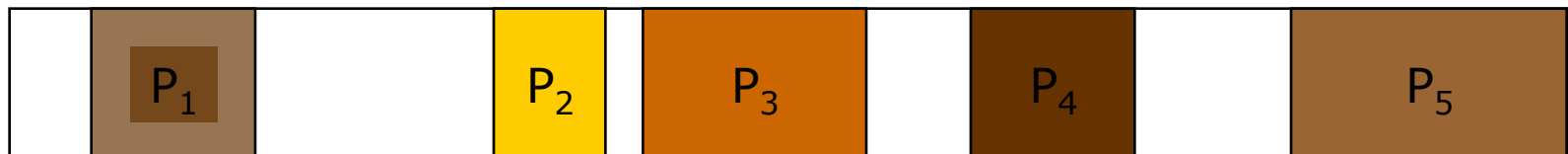
□ הרחבה של השיטה הקודמת, על-ידי תוספת רגיסטר המציין את אורך החתיכה.

□ מונע שיברור פנימי...

□ כמה מקום להקצות לתהליך שמגיע?

שיקרוז חיצוני

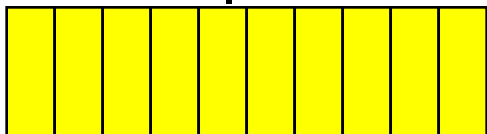
שאריות מקום בין החתיכות שלא מתאימות לכלום.



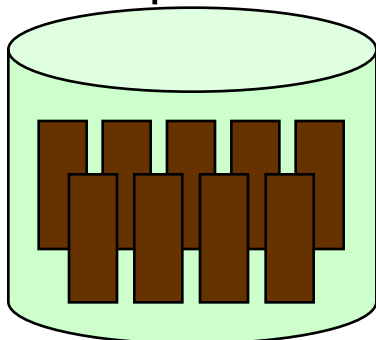
external fragmentation

שיטה מודרנית: דפדוף

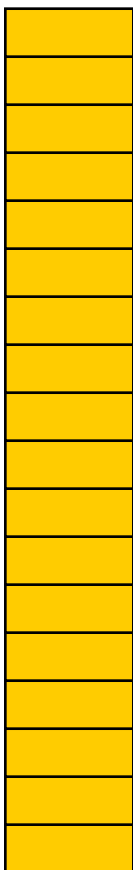
זיכרון פיזי



דיסק

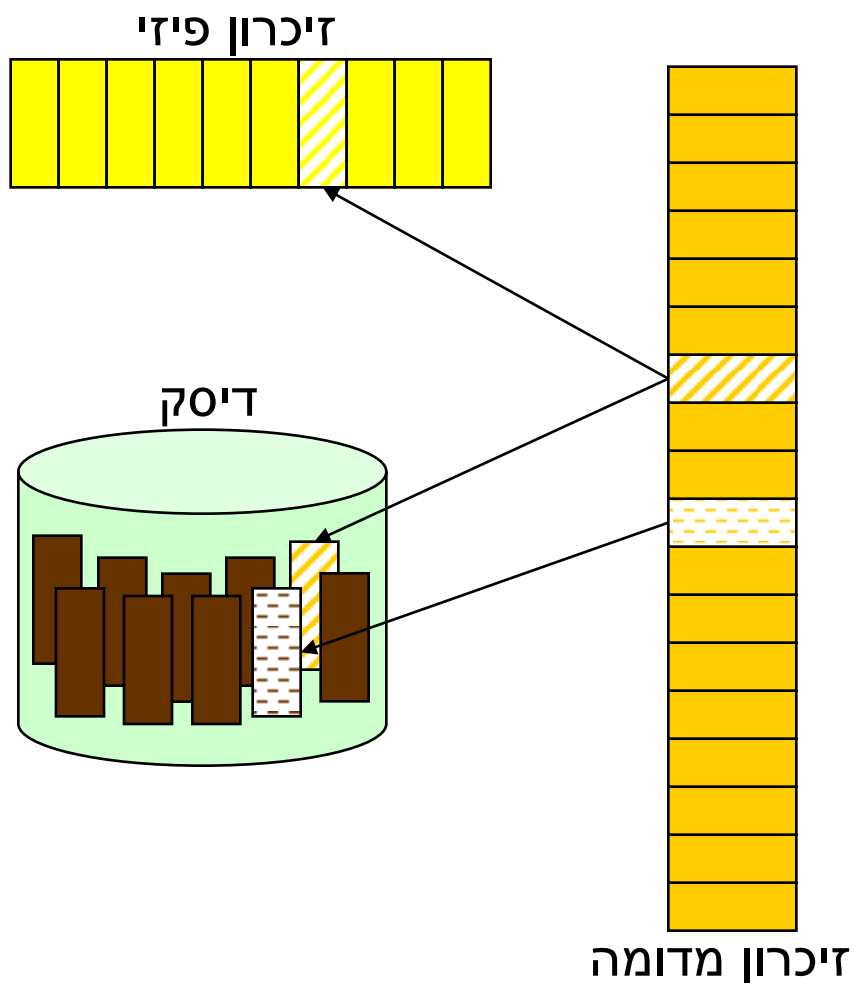


זיכרון מדומה



- מחלקים את הזיכרון הוירטואלי ל**דפים** בגודל קבוע (pages).
- גדולים מספיק לאפשר כתיבה / קריאה יעילה לדיסק.
- קטנים מספיק לתת גמישות.
- גודל טיפוסי = 4K.
- הזיכרון הפיזי מחולק ל**מסגרות** (frames) בגודל דף

דפדוף

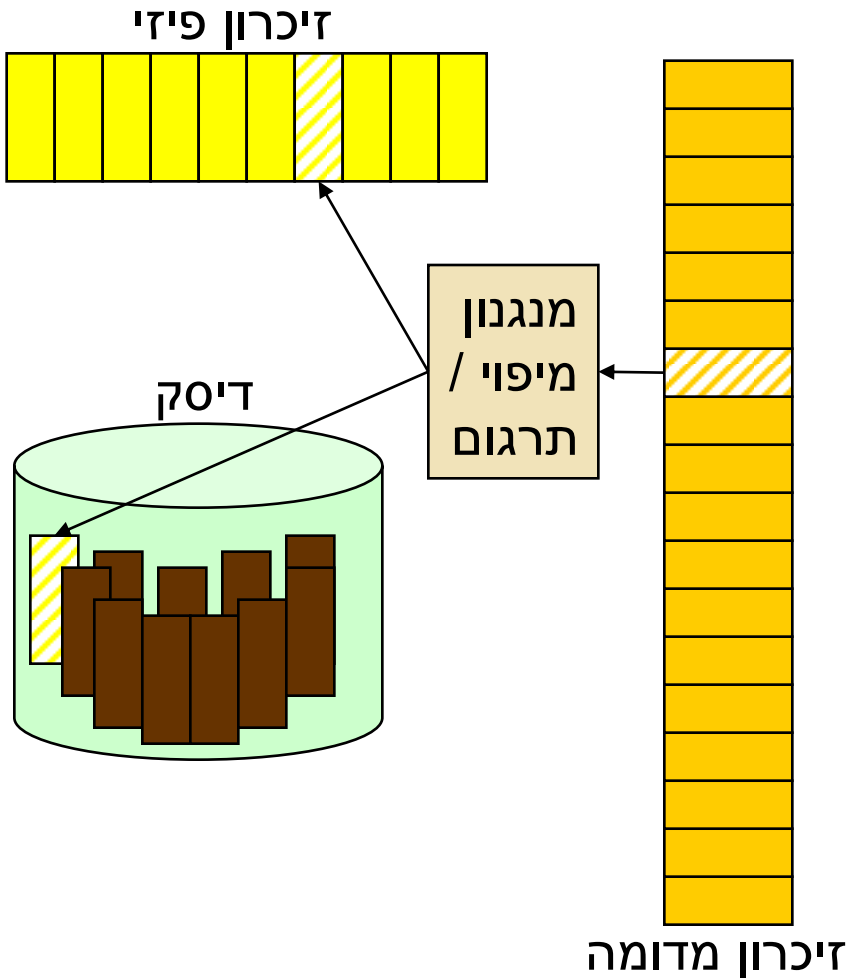


□ כל מסגרת בזיכרון הפיזי יכולה להחזיק כל דף וירטואלי.

□ כל דף וירטואלי נמצא בדיסק.

□ חלק מהדפים הוירטואליים נמצאים בזיכרון הפיזי.

אנאון התראות



□ צריך למפות מכתובת וירטואלית לכתובת פיזית (בזיכרון הראשי או בדיסק).

■ איזה דף נמצא איפה?

■ ומהר...

■ דורש תמיכת חומרה.

טבלת הדפים

□ אחת לכל תהליך.

□ כל כניסה בטבלת הדפים מתייחסת למספר דף וירטואלי

■ זהו האינדקס של הכניסה.

ומכילה מספר מסגרת פיזית

■ זהו ערך הכניסה.

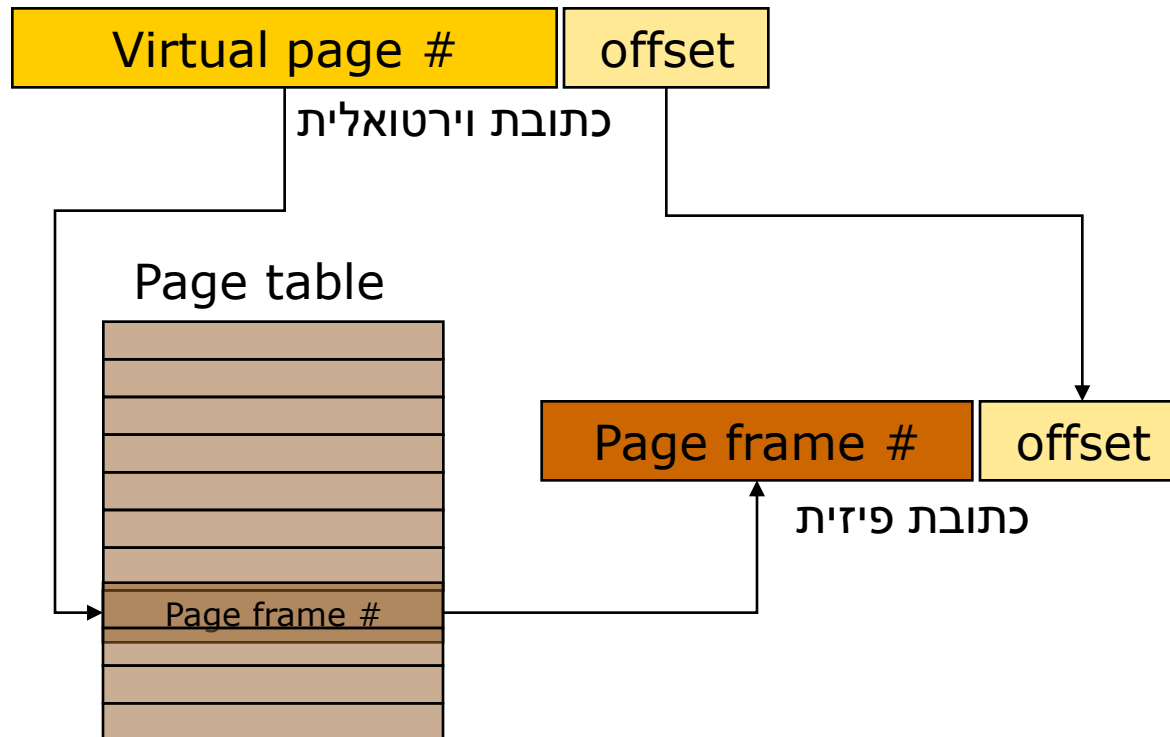
□ לכתובת יש שני חלקים:

■ מספר הדף (Virtual Page Number).

■ offset (מיקום בתוך הדף).

מיפוי

□ ה VPN מהווה מצביע לטבלת הדפים ומאפשר למצוא את מספר המסגרת שבו נמצא הדף (Physical Page Number).



קנדי

- כתובת וירטואלית של 32-ביט.
- מרחב הכתובות מכיל 2^{32} כתובות (4GB=).
- דפים עם 4K ($= 2^{12}$) כתובות.
- 12 ביטים ל- offset.
- 20 ביטים ל- VPN.

00000000011100000000110000000000

כתובת וירטואלית

00000000100100000000110000000000

תתורגם ל

כניסות בטבלת הדפים

מכילות גם מידע ניהולי, בנוסף למיקום הדף הפיזי:



valid bit: האם הכניסה רלוונטית. □

■ מודלק כאשר הדף בזיכרון, כבוי אם הדף נמצא רק בדיסק.

reference bit: האם ניגשו לדף. □

■ מודלק בכל גישה לדף.

modify bit: האם הייתה כתיבה לדף. □

■ בהתחלה כבוי, מודלק כאשר יש כתיבה לדף.

protection bits: מה מותר לעשות על הדף. □

Page Fault

□ כאשר החומרה ניגשת לזיכרון לפי טבלת הדפים ומגיעה לדף שאינו בזיכרון הפיזי, נגרמת חריגה מסוג

page fault

■ בטיפול בחריגה זו, גרעין מערכת ההפעלה טוען את הדף המבוקש למסגרת בזיכרון הפיזי ומעדכן טבלאות דפים

□ ייתכן שיהיה צורך לפנות דף ממסגרת בזיכרון לצורך טעינת הדף החדש ... כולל כתיבת הדף הישן לדיסק אם הוא עודכן

■ כאשר מסתיים הטיפול בחריגה, מבוצעת ההוראה מחדש
restartable instruction □

■ Page Fault יכולה להגרם גם מסיבות אחרות: גישה לא חוקית לזיכרון, גישה לדף לא מוקצה ועוד

דפדוף: הטוב

□ חוסך שיברור חיצוני:

- כל מסגרת פיזית יכולה לשמש לכל דף וירטואלי.
- מערכת-ההפעלה זוכרת איזה מסגרות פנויות.

□ מצמצם שיברור פנימי:

- דפים קטנים בהרבה מחתיכות.

□ קל לשלוח דפים לדיסק:

- בוחרים גודל דף שמתאים להעברה בבת-אחת לדיסק.
 - לא חייבים למחוק, ניתן רק לסמן כלא-רלוונטי (ביט V).
- ## □ אפשר להחזיק בזיכרון הפיזי קטעים לא-רציפים.

דפדוף: הרצ

□ גודל טבלאות הדפים:

■ 4 בתים לכל כניסה, 2^{20} כניסות

← טבלת דפים בגודל 4MB לכל תהליך.

■ ואם יש 20 תהליכים???

□ תקורה של גישות לזיכרון.

■ לפחות גישה נוספת לזיכרון (לטבלת הדפים) על מנת לתרגם את הכתובת.

□ עדיין יש שברור פנימי:

■ גודל זיכרון התהליך אינו כפולה של גודל הדף (שאריות בסוף).

■ דפים קטנים ממזערים שברור פנימי, אבל דפים גדולים מחפים על שהות בגישה לדיסק (disk latency).

טבלת דפים בשתי-רמות

□ חוזרים על אותו תעלול, ושולחים חלקים מטבלת הדפים לדיסק.

■ למשל, דפים של טבלת הדפים שמתייחסים לתהליכים לא פעילים.

■ רמה נוספת של הצבעה.

□ טבלת דפים **כריכה** מאפשרת למצוא את הדפים של טבלת הדפים.

חלוקת הכתובת בשתי-רמות

□ לכתובת וירטואלית יש שלושה חלקים:

Virtual page #		offset
master page #	secondary page #	offset

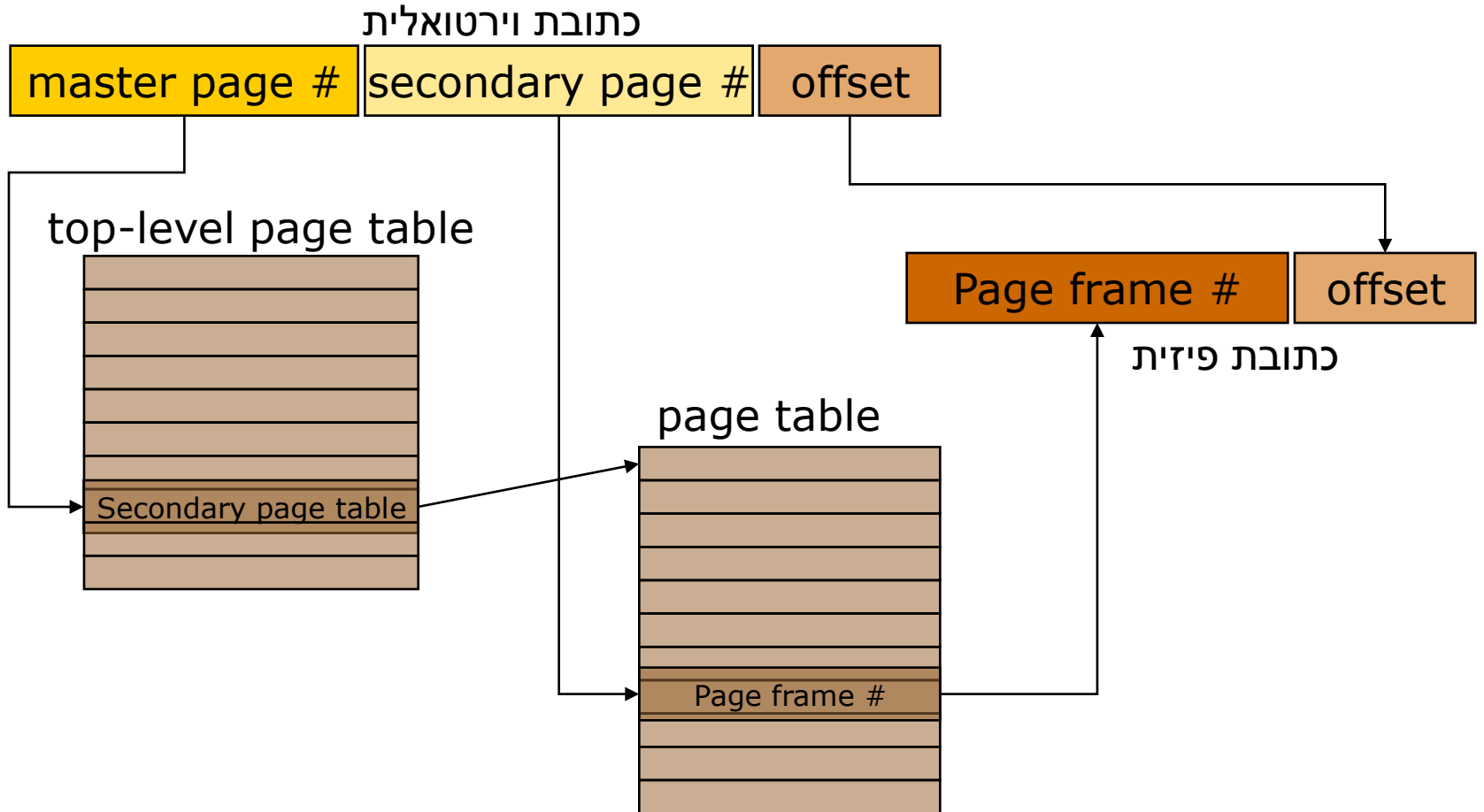
□ רצוי שהטבלה ברמה העליונה תכנס בדף אחד

■ $1024 = 4\text{KB}$ כניסות כל-אחת עם 4 בטים.

■ החלק העליון של הכתובת צריך להיות עם 10 ביטים.

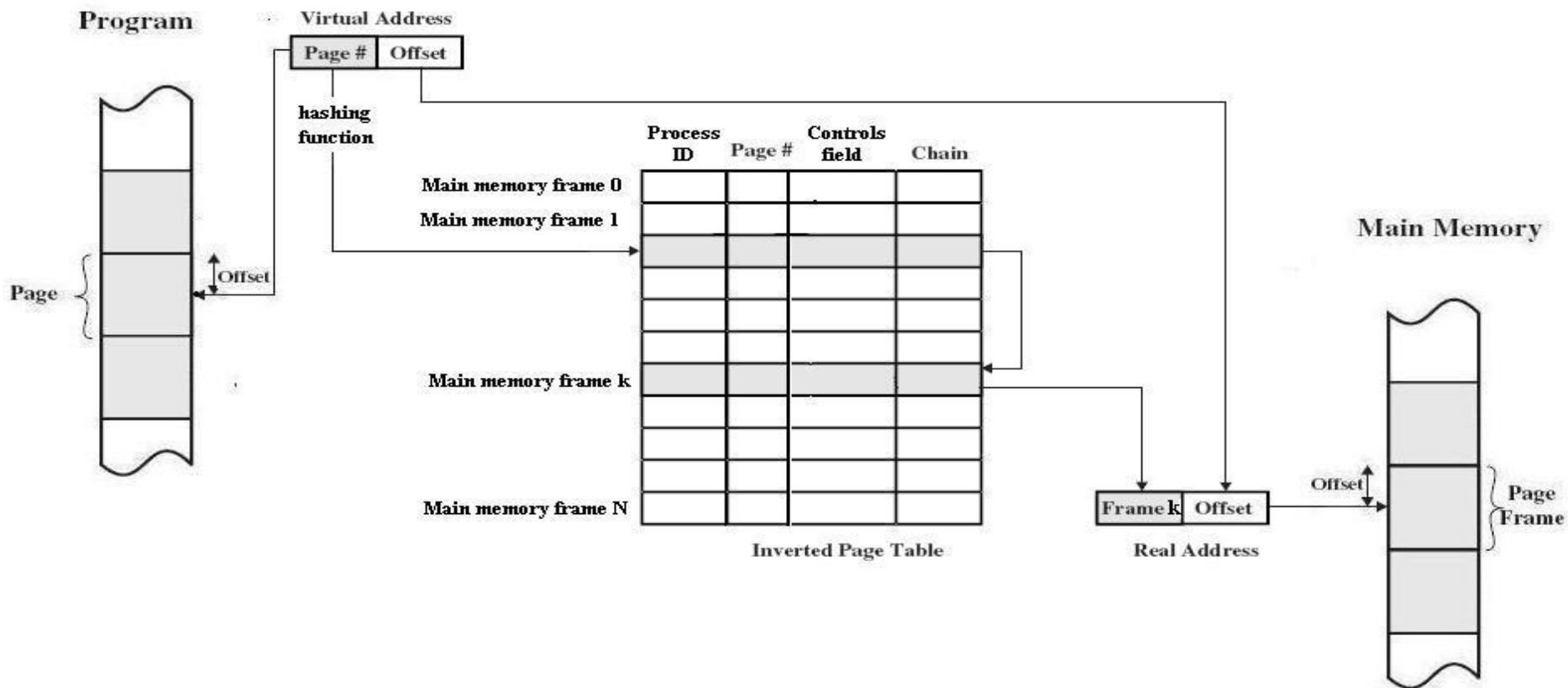
■ גם החלק התחתון 10 ביטים.

טעלע דעם קעגן-נאכט: אינאיינעם



טבלת דפים מהופכת *inverted page table*

השיטה עושה שימוש בטבלת דפים מהופכת שמכילה מיפוי של מסגרות זיכרון לדפים של תהליכים או, במילים אחרות, כל שורה בטבלת דפים מהופכת מתייחסת למסגרת בזיכרון הפיזי ומכילה אינפורמציה לגבי איזה דף ממופה במסגרת ולאיזה תהליך הוא שייך. חיפוס בטבלה נעשה בעזרת פונקצית ערבול (HASH) שמראה איפה דף רצוי יכול להימצא לפי סדר עדיפויות. אם הוא לא שם-עוד פעם מחשבים את הפונקציה עד שמוצאים או אין יותר אפשרויות ואז זה אומר שהוא בדיסק. שימוש בטבלת זיכרון מהופכת חוסך כמות גדולה של זיכרון, ולכן ניתן להחזיק אותה ברמות גבוהות בהיררכיה של זיכרון.




דפדוף e טבלת הדפים

- הטבלה ברמה העליונה תמיד בזיכרון הראשי.
- תקורה של שתי גישות זיכרון (ולפעמים גישה לדיסק!) על כל גישה לדף.

פתרון?

שימוש במטמון (cache) מיוחד.

קטן וזריז...

Babylon English-Hebrew 

cache •
(פ') להחביא; להסמין

(ש"ט) מטמון (גם במחשבים); (במחשבים) אזור
אחסון המכיל נתונים שלהם יזדקק המחשב תוך זמן
קצר, מחבוא

Translation Lookaside Buffer (TLB)

- מטמון חומרה אשר שומר מיפויים (=תירגומים) של מספרי דפים וירטואליים למספרי דפים פיזיים.
- למעשה, שומר עבור כל מספר דף וירטואלי את כל הכניסה שלו בטבלת הדפים, אשר יכולה להכיל מידע נוסף (זהו הערך).
- המפתח הוא מספר הדף הוירטואלי.
- קטן מאוד: 16-48 כניסות (סך-הכל 64-192KB).
- מהיר מאוד: חיפוש במקביל (תוך cycle אחד או שניים).
- אם כתובת נמצאת ב TLB (פגיעה, **hit**) ← הרווחנו.
- אם יש החמצה (**miss**) ← תרגום רגיל דרך טבלת הדפים.

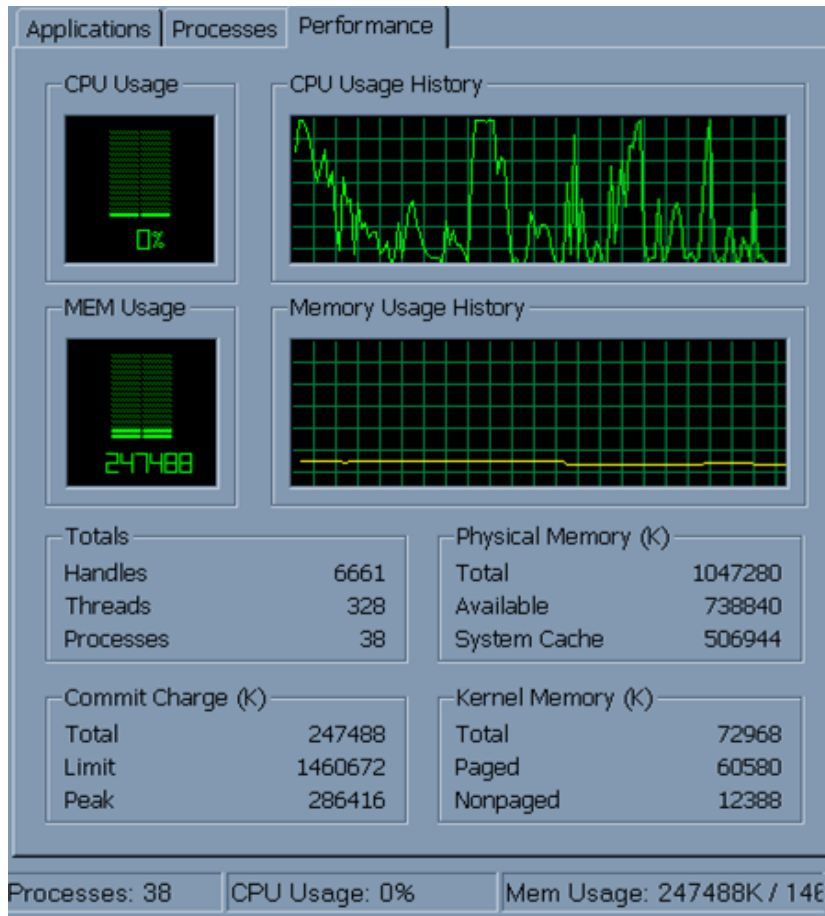
ביצועי TLB

- מאפשר תרגום עבור 64-192KB כתובות זיכרון.
- אם תהליך ניגש ליותר זיכרון ← יותר החטאות ב TLB.

■ פגיעות כמעט ב 99% של הכתובות!

- מידה רבה של מקומיות (**locality**) בגישה לזיכרון.
- לוקליות במקום (spatial locality): מסתובבים בכתובות קרובות.
(אם ניגשתי לכתובת x , סיכוי טוב שאגש לכתובת $x+d$).
- לוקליות בזמן (temporal locality): חוזרים לאותן כתובות בזמנים קרובים.
(אם ניגשתי לכתובת x בזמן t , סיכוי טוב שאגש אליה גם בזמן $t+\Delta$).

כמה זיכרון פיזי תהליך צריך באמת?



□ לתהליכים שונים, דרישות זיכרון שונות.

□ אם תהליך משתמש באופן פעיל בהרבה זיכרון, אין מה להריץ אותו כאשר הזיכרון הפיזי מלא מידי.

Working set

□ קבוצת העבודה של תהליך P , $WS_P(w)$,
 = הדפים אליהם תהליך P ניגש ב w הגישות האחרונות.

זמן	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
דף #	2	6	1	5	7	7	7	7	5	1	6	2	3	4	4	4	3
$WS(3)$	2	2 6	2 6 1	6 1 5	1 5 7	5 7	7	7	5 7	5 7 1	5 1 6	2 6 1	2 6 3	2 4 3	4 3	4	4 3

- ככל שהקבוצה קטנה יותר (עבור ערך w קבוע) יש יותר מקומיות בגישות לזיכרון של התהליך.
- אם קבוצת העבודה לא נמצאת בזיכרון, מערכת ההפעלה מתמוטטת מרוב הבאות / פינויים של דפים (**thrashing**).

דפדוף לפי דרישה

□ מביאים דף רק אם התהליך דורש אותו (demand paging)

■ למשל, בהתחלת הביצוע ניגשים לדפים חדשים, של נתונים ושל קוד.

■ אם הדף לא נמצא בזיכרון, קורה page fault.

□ לעומת זאת, **prepaging** מנסה לנבא איזה דפים ידרשו, ומביא אותם מראש.

■ גישה לדיסק תוך כדי חישוב אחר במעבד.

מדיניות ההחלפה

את מי מפנים?

■ מטרה עיקרית: מזעור מספר פסיקות הדף.

■ מחיר פינוי דף "מלוכלך" יקר יותר מאשר פינוי דף נקי.

■ מתי עושים מה:

■ טיפול בפסיקת דף מתבצע on-line.

■ תהליך פינוי דפים מתבצע בדרך-כלל ברקע (off-line), כאשר מספר המסגרות הפנויות יורד מתחת לאיזשהו סף.

■ לפי עקרון "סימני המים" (water-marks).

■ דפים מלוכלכים נכתבים ברקע לדיסק.

FIFO אלקוריתם

מפנה את הדף שנטען לפני הכי הרבה זמן

	A	B	C	D	A	B	C	D	A	B	C	D
0	A	A	A	D	D	D	C	C	C	B	B	B
1		B	B	B	A	A	A	D	D	D	C	C
2			C	C	C	B	B	B	A	A	A	D

Belady אנונייט

FIFO באדאריתם

	A	B	C	D	A	B	E	A	B	C	D	E
0	A	A	A	D	D	D	E	E	E	E	E	E
1		B	B	B	A	A	A	A	A	C	C	C
2			C	C	C	B	B	B	B	B	D	D

0	A	A	A	A	A	A	E	E	E	E	D	D
1		B	B	B	B	B	B	A	A	A	A	E
2			C	C	C	C	C	C	B	B	B	B
3				D	D	D	D	D	D	C	C	C

אלגוריתם אופטימלי

□ אם כל הגישות לזיכרון ידועות מראש...

□ האלגוריתם **החמדן** מפנה מהזיכרון את הדף שהזמן עד לגישה הבאה אליו הוא הארוך ביותר.

	A	B	C	A	B	D	A	D	B	C	B
0	A	A	A	A	A	A	A	A	A	C	C
1		B	B	B	B	B	B	B	B	B	B
2			C	C	C	D	D	D	D	D	D

Least Recently Used (LRU)

מפנה את הדף שהגישה האחרונה אליו היא המוקדמת ביותר

	A	B	C	A	B	D	A	D	B	C	B
0	A	A	A	A	A	A	A	A	A	C	C
1		B	B	B	B	B	B	B	B	B	B
2			C	C	C	D	D	D	D	D	D

מתנהג כמו
החמדן

	A	B	C	D	A	B	C	D	A	B	C	D
0	A	A	A	D	D	D	C	C	C	B	B	B
1		B	B	B	A	A	A	D	D	D	C	C
2			C	C	C	B	B	B	A	A	A	D

מתנהג כמו
FIFO

LRU מ'א'ל'ש

□ חותמת זמן (timestamp) לכל דף

■ בגישה לדף מעדכנים את החותמת

■ מפנים את הדף עם הזמן המוקדם יותר

□ ניהול מחסנית

■ בגישה לדף מעבירים את הדף לראש המחסנית

■ מפנים את הדף בתחתית המחסנית

□ יקר ודורש תמיכה בחומרה

LRU מקורב:

אלגוריתם ההלדמנות השנייה

reference bit לכל דף.

□ בגישה לדף, הביט מודלק.

□ בפינוי דפים, מערכת ההפעלה עוברת באופן מחזורי על כל הדפים

■ אם $\text{reference bit} = 0$, הדף מפונה.

■ אם $\text{reference bit} = 1$, מאפסים את הביט.

LRU מקור: שיטה טובה

- מצרפים שדה גיל (age) לכניסה של דף בטבלת הדפים.
- בגישה לדף מבצעים:
 $R = 1$
כל יחידת timer, עבור כל דף בזיכרון הפיזי מבצעים:
`if (R) then`
 `age = 0`
 `R = 0`
`else`
 `age++`
 `if (age > threshold)`
 `evict page`
- דפים שלא ניגשו אליהם הרבה זמן, מפונים מהזיכרון.

מדיניות דפדוף ב Windows NT

- לפי דרישה, אבל מביאים קבוצת דפים מסביב.
- אלגוריתם הדפדוף הוא FIFO על הדפים של כל תהליך בנפרד.
- מנהל הזיכרון הוירטואלי עוקב אחרי ה-working set של התהליכים, ומריץ רק תהליכים שיש בזיכרון מקום לכל ה-working set שלהם.
- גודל ה-working set נקבע עם התחלת התהליך, לפי בקשתו ולפי הצפיפות הנוכחית בזיכרון.

מדיניות דפדוף ב Linux (2.4.X)

- גם כאן לפי דרישה, ומביאים קבוצת דפים מסביב
 - דפים סמוכים מאוחסנים יחד בדיסק ונטענים יחד כשאחד מהם נדרש
- אלגוריתם הדפדוף הוא גלובלי - קירוב ל-LRU
- שלוש הזדמנויות + ניהול מחסנית
 - כל גישה נותנת הזדמנות אחת נוספת.
 - דומה ל-LFU (Least Frequently Used)
 - כל הדפים בזיכרון הפיזי נמצאים במחסנית, ממוינת לפי מספר ההזדמנויות.
 - דף שנטען לזיכרון מתחיל באיזור 0 הזדמנויות ועובר מיד לאיזור של 1 (כשניגשים אליו).
 - פינוי דפים מבוצע מתחתית המחסנית

חדיניות דפדוף ב Linux (2.4.X)

□ הדיסק מופעל כאמצעי אחרון בלבד

■ דפים מפונים רק כאשר הזיכרון הפיזי מתמלא מעבר לסף העליון

...ואז מפנים עד מתחת לסף תחתון

□ הזיכרון המשמש את הגרעין

■ אינו מדופדף כלל לדיסק

■ ממופה לחלק של הזיכרון הוירטואלי אשר משותף לכל התהליכים ("הג'יגהבייט הרביעי")

סמנטיקה

□ חלוקה של זיכרון התהליך לחלקים עם משמעות סמנטית.

■ קוד, מחסנית, משתנים גלובליים...

■ בגודל שונה (⇔ יותר שיברור חיצוני).

■ הביטים העליונים של הכתובת הוירטואלית מציינים את מספר הסגמנט.

■ ניהול כמו זיכרון עם חלוקה משתנה (חתיכות באורך לא-קבוע).

■ לכל סגמנט, רגיסטר בסיס וגבול, המאוחסנים בטבלת סגמנטים.

■ מדיניות שיתוף והגנה שונה לסגמנטים שונים.

□ אפשר לשתף קוד, אסור לשתף מחסנית זמן-ביצוע.

□ שילוב עם דפדוף.

דואל: Linux בסביבת IA32

- מערכות IA32 מאפשרות שילוב של סגמנטציה ודפדוף
 - סגמנט מוגדר כאוסף רציף של דפים וירטואליים
- Windows מנצלת תמיכה זו וממש יוצרת לכל תהליך סגמנט נפרד לקוד, סגמנט נפרד לנתונים וכו'
- Linux אינה מנצלת את מנגנון הסגמנטציה המוצע בחומרה
 - עובדת גם על ארכיטקטורות חומרה שתומכות בדפדוף אבל לא בסגמנטציה
- עם זאת, Linux מכילה מנגנון סגמנטציה פנימי בצורה של **איזורי זיכרון**
 - דפדוף של כל הסגמנטים עם טבלת דפים עם **שלוש-רמות**.

מערכת הקבצים

- מבוא: מטרות מערכת קבצים
- מנשק המשתמש: פעולות על קבצים, ארגון קבצים, הגנה
- תכונות של דיסקים.
- מימושים: בסיסיים וקצת על מימושים מתקדמים.
- אמינות מערכת הקבצים.

מצרכות קבצים

□ קבצים מאפשרים שמירת מידע לטווח בינוני וארוך.

□ למרות הפעלות חוזרות של תוכנית, איתחולים, ונפילות.

□ מטרות:

■ הפשטה מתכונות ספציפיות של ההתקן הפיזי

□ גישה דומה לדיסק, CD-ROM, DVD, טייפ, ...

■ זמן ביצוע סביר.

■ ארגון נוח של הנתונים.

■ הפרדה בין משתמשים שונים (protection).

■ הגנה (security).

מבנה פואי של מערכת קבצים

□ אוסף של קבצים.

□ קובץ הוא מידע עם תכונות שמנוהלות על-ידי המערכת.

■ מכיל מספר בתיים/שורות/רשומות בגודל קבוע/משתנה.

■ קובץ יכול להיות מסוג מסוים

□ חלק מהסוגים מזהים על-ידי המערכת: מדרך, link, mount

□ ...או על-ידי אפליקציות: .exe, .doc, .html, .jpg.

□ במקביל, ניתן להתייחס לתכולת הקובץ: בינארי או טקסט

תכונות *ef* קובץ (Attributes)

□ תכונות מערכת

■ שם

□ תווים מותרים? upper / lower case.

■ גודל, בדרך-כלל בבתים

■ מיקום.

■ מידע על בעלות והגנה.

■ תוויות זמן: יצירה, גישה אחרונה, שינוי אחרון.

□ תכונות משתמש

■ ב-HPFS (מערכת הקבצים של OS/2)

ניתן להצמיד לקובץ זוגות $\langle \text{attribute_name}, \text{string} \rangle$
(מחרוזת באורך עד 2GB).

פעולות בסיסיות על קבצים

□ יצירה/השמדה של קובץ:

- create: מקצה מקום לקובץ, ושומר את התכונות שלו.
- delete: משחרר את המקום שהוקצה לקובץ, וכן את התכונות שלו.

□ קריאה/כתיבה:

- read: נותן שם קובץ, כמות מידע לקריאה וחוצץ זיכרון שבו יאוחסן המידע הנקרא; מתחזק מצביע מיקום לקובץ.
- write: נותן שם קובץ ומידע לכתיבה; מתחזק מצביע מיקום לקובץ.
- seek: הזזת מצביע המיקום.

פעולות נוספות על קבצים

open, close □

■ מאחזרות מידע על הקובץ, מאתחלות את מצביע המיקום לקובץ.

■ משפרות את ביצועי המערכת.

append, rename, copy, ... □

■ ניתן לממשן בעזרת פקודות אחרות...

□ נעילה.

□ עדכון תכונות הקובץ

אופני גישה לקובץ

□ גישה **סדרתית**: ניגשים למידע בקובץ לפי סדר.

■ בד"כ מהירה יותר.

■ לא צריך לציין מהיכן לקרוא (ניתן להתבסס על מצביע המיקום).

■ מאפשר למערכת להביא נתונים מראש.

□ גישה **ישירה** / אקראית.

■ לפי מיקום או לפי מפתח.

■ לפעמים ניתן לזהות תבניות גישה.

■ כמובן, ניתן לממש פה גם גישה סדרתית.

ארכיון מצרית קבצים

□ מספר הקבצים מגיע לאלפים

■ ישנן מערכות שמכילות מיליארדי קבצים!

□ **מחיצות** (partitions), בד"כ לפי התקנים (devices).

□ **מדריכים** (directories), טבלאות הקבצים בתוך המחיצה.

מדרכים

□ המדריך הוא מבנה נתונים מופשט, אשר שומר את התכונות של כל קובץ הנמצא בו.

□ תומך בפעולות הבאות:

- מציאת קובץ (לפי שם)
- יצירת כניסה
- מחיקת כניסה
- קבלת רשימת הקבצים בתוך המדריך
- החלפת שם של קובץ

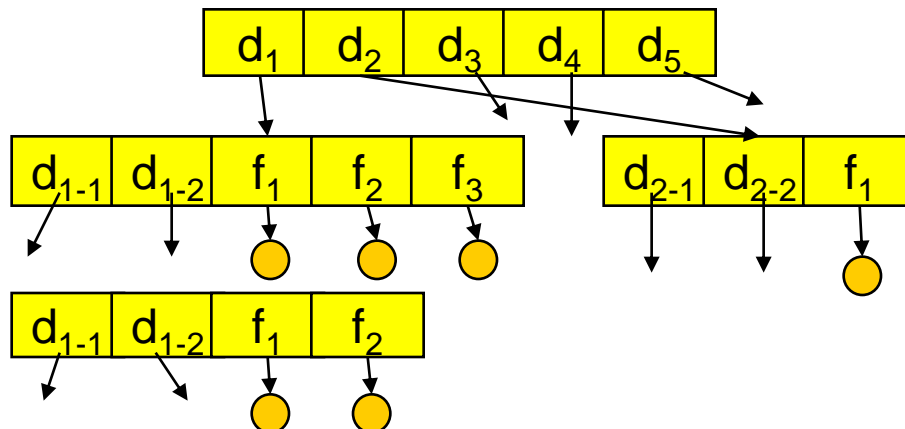
...

מבנה מדריכים: עץ מכוון

□ עץ בעל מספר רמות שרירותי (MS-DOS)

■ מושג **המדריך הנוכחי** (current directory).

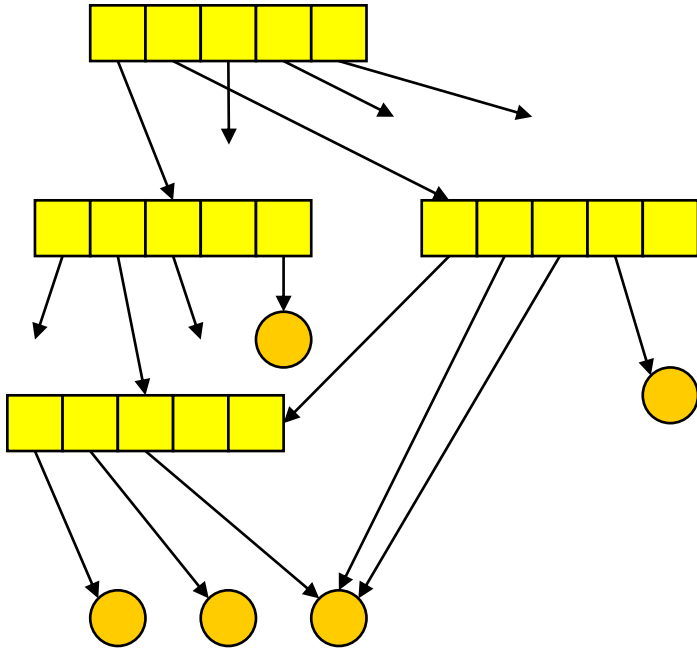
■ קובץ מזוהה ע"י מסלול מהשורש (מוחלט) או מהמדריך הנוכחי (יחסי).



מבנה מדרגי: ארץ אציקלי

□ בגרף אציקלי, מאפשרים
למספר כניסות במדרגים (אולי
שונים) להצביע לאותם העצמים
(Unix)

□ למה לא לאפשר מעגלים בגרף?



סוגי קישורים

□ קישור רך / סימבולי (soft / symbolic link):
כניסה שמכילה את המסלול לקובץ.

□ קישור חזק (hard link):
לא ניתן להבדיל מהכניסה המקורית

□ מחיקת קובץ:

■ קישור סימבולי אינו מונע מחיקת הקובץ

□ משאיר מצביע תלוי באוויר

■ קישור חזק מחייב מחיקת כל הכניסות.

מבנה מדיכיס Unix-ק

- שם (מסלול מלא) אינו תכונה של קובץ...
 - לאותו קובץ ניתן להגיע דרך מסלולים שונים
- מחזיקים use-counter שמאפשר לדעת מתי למחוק קובץ.
 - תהליך יכול ליצור קובץ, לפתוח אותו, למחוק את הכניסה (היחידה) שלו מהמדריך ולהמשיך לעבוד עליו
- לתהליך מבנה נתונים של קבצים פתוחים.
 - בד"כ כולל מידע על מיקום נוכחי בקובץ שאינו משותף עם פתיחות אחרות של אותו קובץ, כולל של אותו תהליך.

האנה

□ רוצים למנוע ממשתמשים לבצע פעולות ספציפיות על הקובץ (בשוגג או במזיד)

■ למשל קריאה, כתיבה, ביצוע, שינוי שם, ...

□ רשימות גישה (access list).

■ לכל קובץ, נרשום למי מותר לבצע איזה פעולה.

■ הרשימות עלולות להפוך לארוכות מדי וקשות לתחזוקה.

■ פיתרון:

□ נקבץ משתמשים למחלקות שונות: owner, group, universe...

□ מספר מוגבל של פעולות: read, write, execute

נציגות קבצים

advisory locking □

- תהליך צריך לנעול קובץ בצורה מפורשת (כמו קטע קריטי)
- אפשר לנעול קובץ שלם או חלקים מהקובץ

mandatory locking □

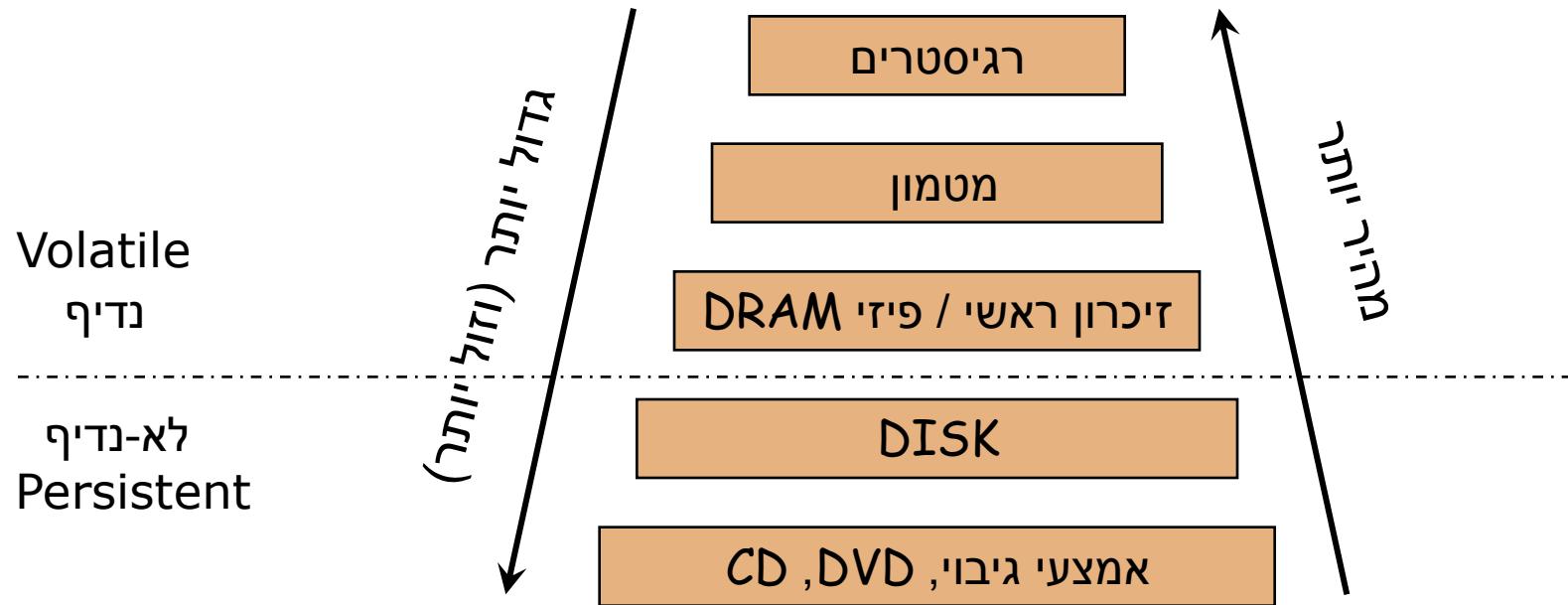
- אם הקובץ נעול – כל תהליך שמבצע read, write, open יינעל, גם אם לא ביצע נעילה מפורשת

אפשר להפריד לפי קריאות וכתובות (reader/writer lock)
ב Linux יש גם advisory locking וגם mandatory locking

leases □

- אם תהליך שנעל את הקובץ לא משחרר אותו ולא מאריך את ה lease תוך פרק זמן נתון, המנעול משתחרר
- שימושי אם התהליך שנעל את הקובץ נפל

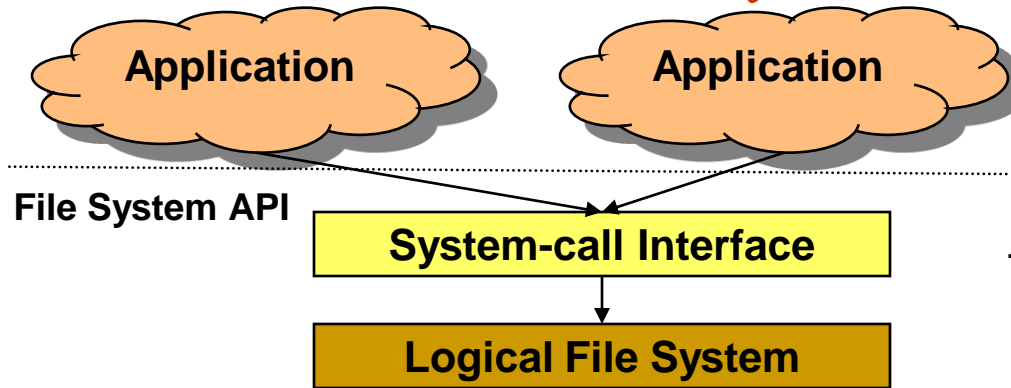
היררכיית האחסון



ליכרון משני

- בניגוד לזיכרון הראשי, אינו מאפשר ביצוע ישיר של פקודות או גישה לבתים.
- שומר על מידע לטווח ארוך (לא-נדיף, non-volatile).
- גדול, זול, ואיטי יותר מזיכרון ראשי.
- נתרכז ב**דיסקים** שהגישה אליהם באמצעות כתיבה או קריאה של רצפי מידע גדולים.
 - גישה סידרתית מהירה.
 - קיבולת גבוהה.

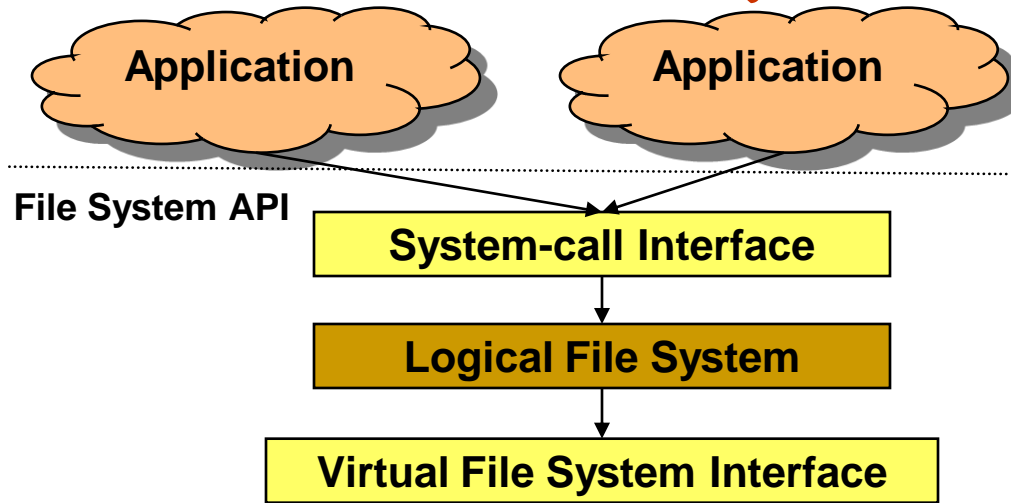
מבנה מערכת קבצים טיפוסית



logical file system □

- קוד כללי, בלתי תלוי במערכת קבצים ספציפית
- קוד שמקבל מסלול, ומחזיר את קובץ היעד
- ניהול מידע כללי עבור קבצים פתוחים, כמו מיקום בקובץ...
- פעולות על מדריכים.
- הגנה ובטיחות.

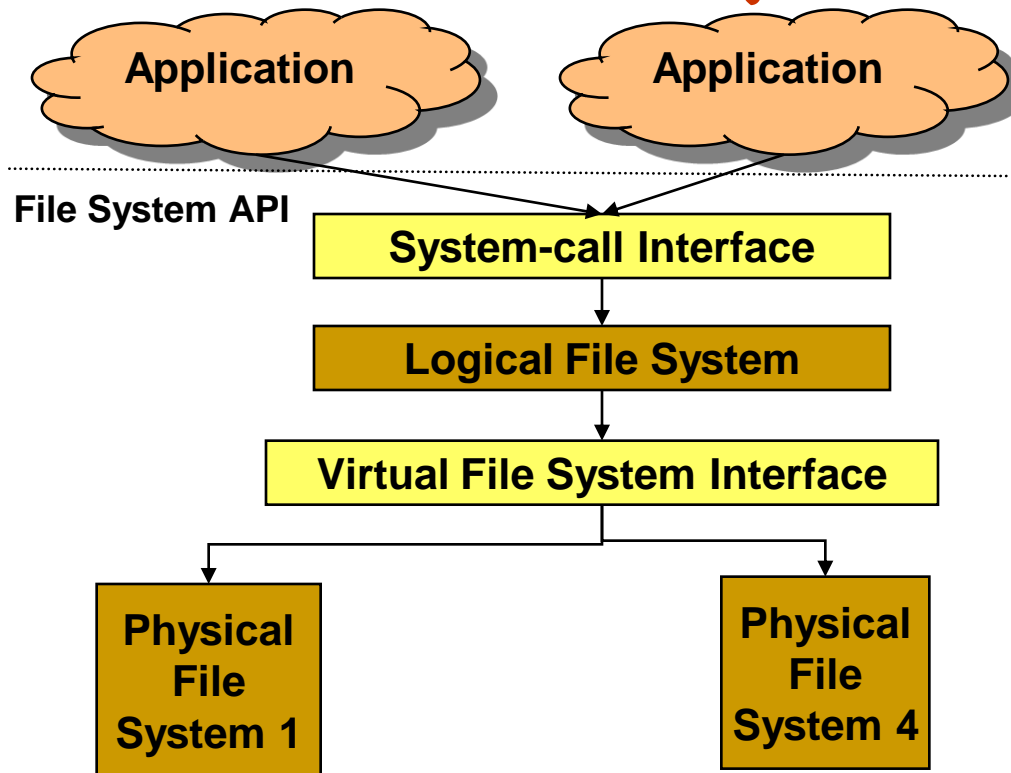
מבנה מערכת קבצים טיפוסית



logical file system □
virtual file system □
interface

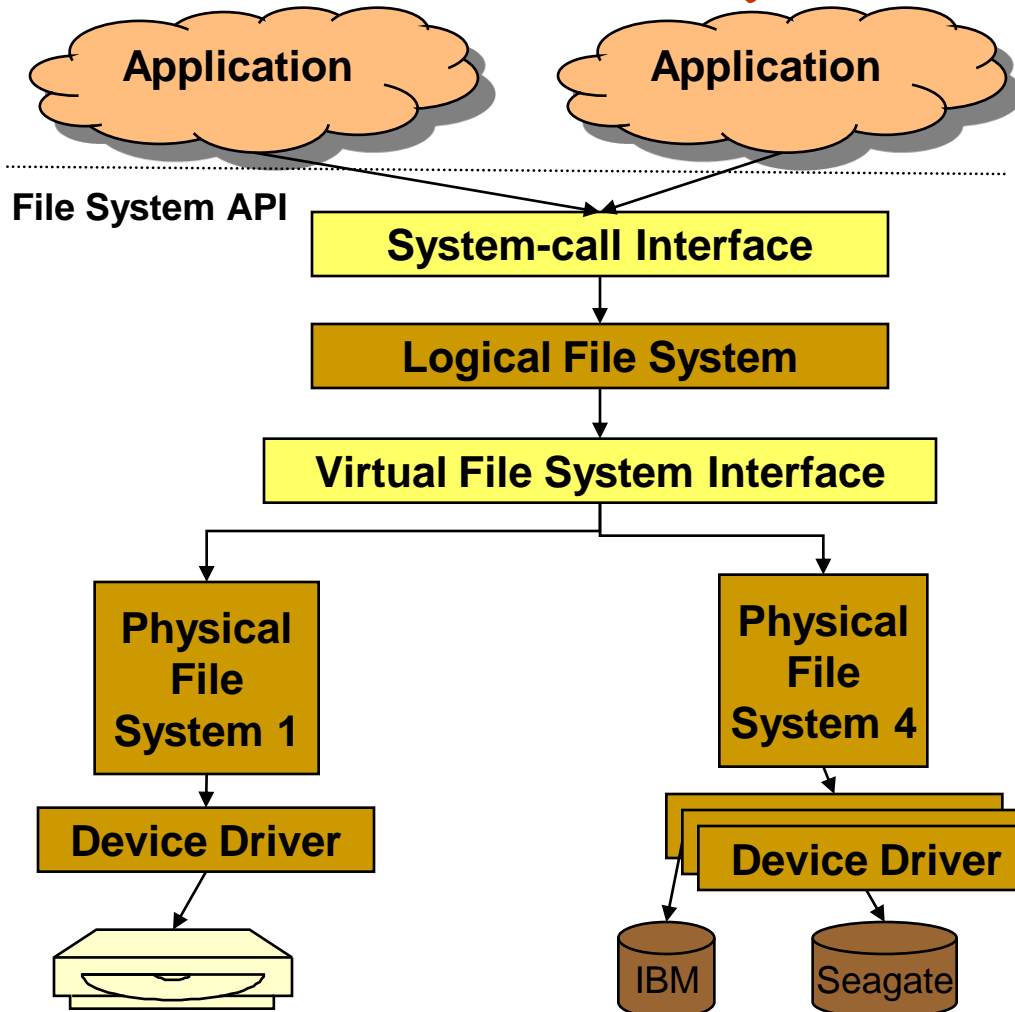
- ממשק אחיד לכל מערכות הקבצים הספציפיות.
- למשל, `vfs_read`, `vfs_write`, `vfs_seek`.

מבנה מערכת קבצים טיפוסית



- logical file system □
- virtual file system □
- interface
- physical file system □
- מימוש ממשק ה-VFS עבור מערכת קבצים ספציפית
- למשל, מעל דיסק, דיסקט, RAM, CD, רשת וכו'
- מתכנן איך לפזר את הבלוקים.
- בהמשך, נתרכז בו.

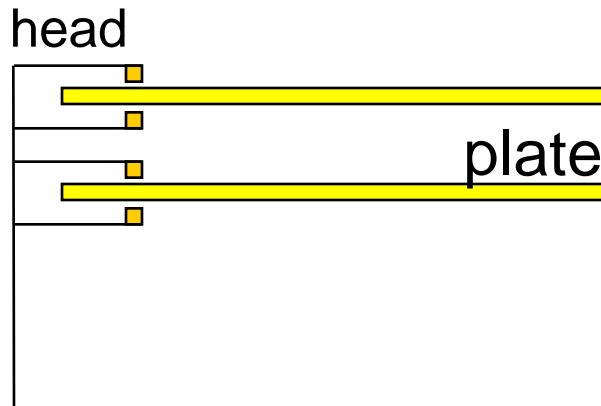
מבנה מערכת קבצים טיפוסית



- logical file system □
- virtual file system □
- interface
- physical file system □
- device drivers □

- קוד שיועד איך לפנות להתקן ספציפי
- דיסקים (לפי מודל), DVD, וכדומה.
- מתחיל את הפעולה הפיזית, ומטפל בסיומה.
- מתזמן את הגישות, על מנת לשפר ביצועים.

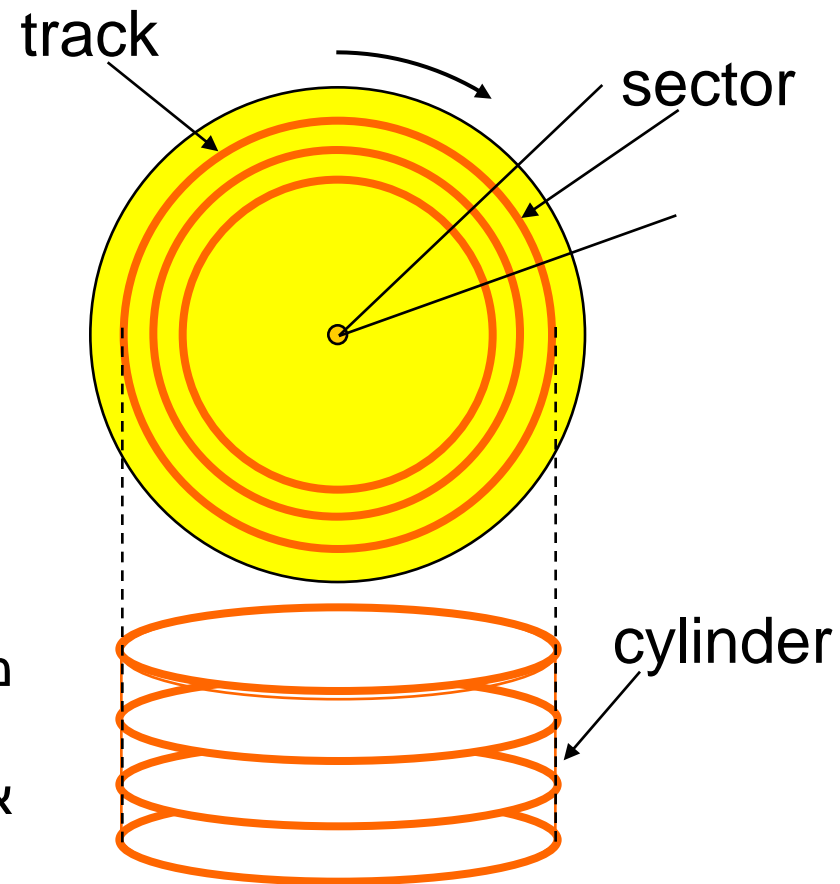
מבנה הקדיסק



מבנה כתובת
(CHS) Cylinder, Head, Sector

(LBA) Logical Block Array

או



דואל: דיסק seagate

עלות - \$1000

r סיבובים לדקה

s / זמן ממוצע מהגעה ל-track עד לסקטור

s זמן ממוצע להזזת זרוע מ-track ל-track

tr קצב העברת נתונים

CHEETAH X15-36LP		36.7 GB
Model Number		ST336752LC/LW/FC
PERFORMANCE		
Spindle Speed (RPM)		15K
Latency, average (msec)		2.0
Seek Time		
Average read/write (msec)		3.6/4.2
Track-to-Track read/write (msec)		0.3/0.4
Transfer Rate		
Internal (Mbits/sec)		522–709

• שימו לב ש $l = \frac{1}{2r}$

• זמן ממוצע לגישה לסקטור אקראי $s + l = 3.6 + 2 = 5.6 \text{ msec}$

• זמן העברה פנימי של סקטור $4KB / tr = 45 \mu \text{sec}$

(IBM) desktop *le kndi? :kosq*

עלות - \$100

$$s + l = 8.5 + 4.17 = 12.67 \text{ msec}$$

IBM Deskstar 120GXP at a glance

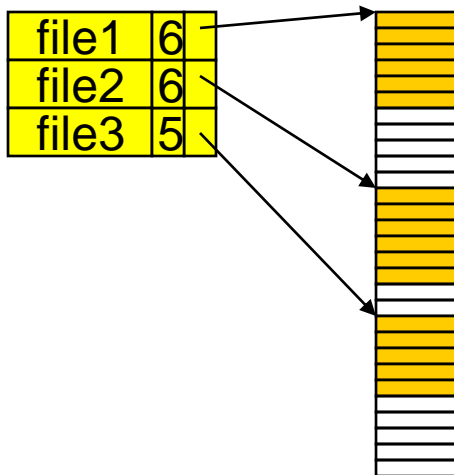
Performance

Data buffer	2MB
Rotational speed (rpm)	7,200
Latency average (ms)	4.17
Media transfer rate (max. Mbits/sec)	592
Interface transfer rate (max. MB/sec)	100
Sustained data rate (MB/sec)	48 to 23
Seek time (read, typical) ³	
Average (ms)	8.5
Track to track (ms)	1.1
Full track (ms)	15

מימוש מצרכת קבצים: מדדים

- מהירות גישה סדרתית
- מהירות גישה אקראית (ישירה)
- שיברור פנימי וחיצוני
- יכולת להגדיל קובץ
- התאוששות משיבושי מידע

מיפוי קבצים: הקצאה רציפה



□ בבלוקים באורך קבוע

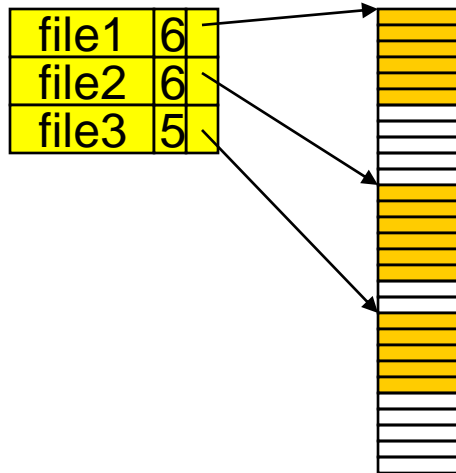
■ כפולה של גודל סקטור
(נע בין 4KB – 512B)

□ המשתמש מזהיר על גודל
הקובץ עם יצירתו.

□ מחפשים בלוקים רצופים
שיכולים להכיל את הקובץ.

□ הכניסה במדריך מצביעה
לבלוק הראשון בקובץ.

מיפוי קבצים: הקצאה רציפה

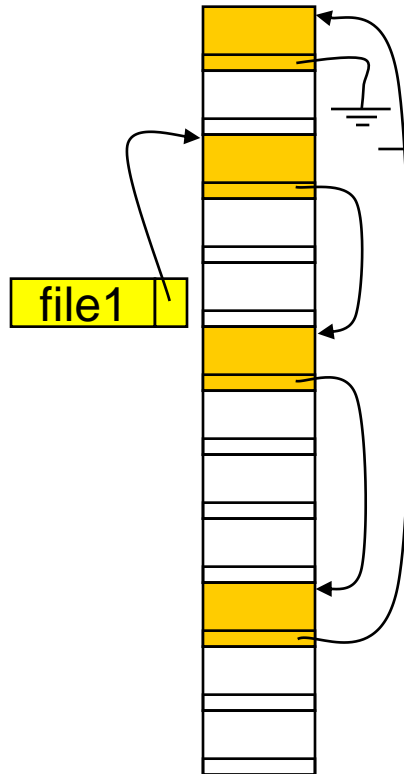


✓ גישה מהירה (סדרתית וישירה)

✗ שיברור פנימי וחיצוני

✗ קשה להגדיל קובץ

מיפוי קבצים: הקצאה משורשרת

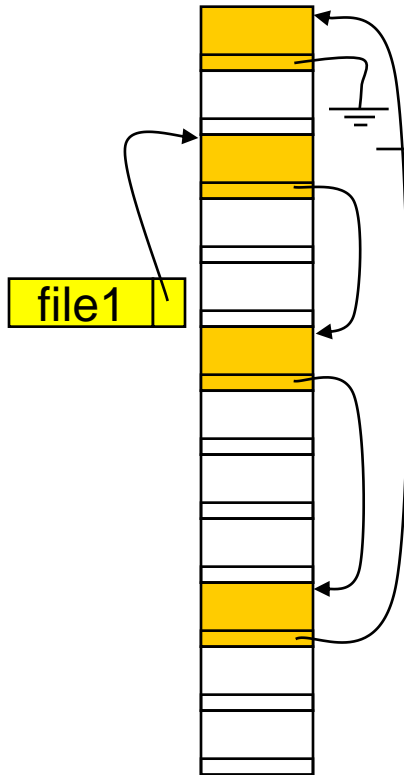


□ כל בלוק מצביע לבלוק הבא.

□ הכניסה במדריך מצביעה

לבלוק הראשון בקובץ

מיפוי קבצים: הקצאה משורשרת



✓ קל להגדיל קובץ.

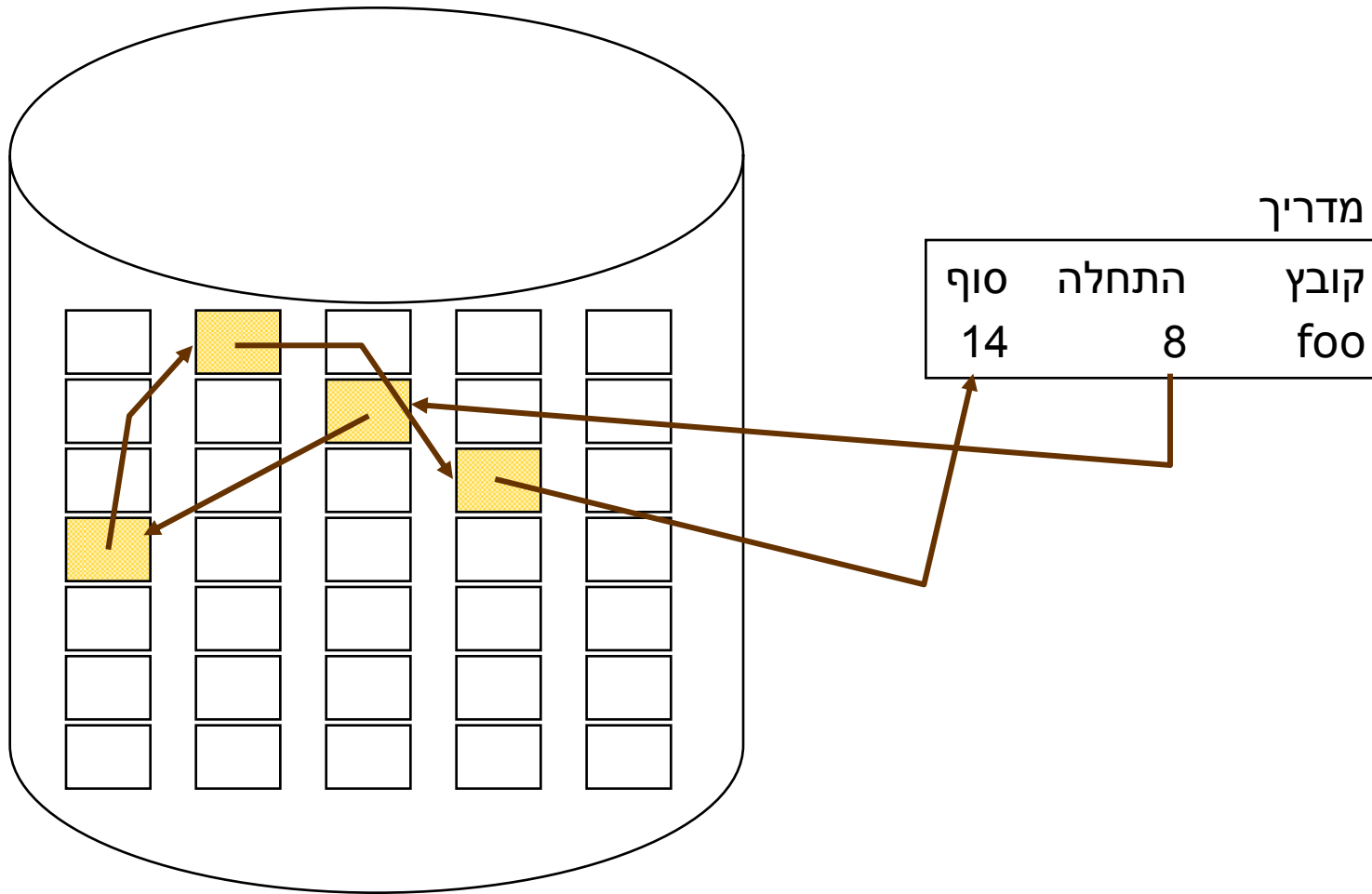
✓ מעט שיברור חיצוני.

✗ גישה איטית, בעיקר לגישה ישירה.

■ שימוש בבלוקים גדולים מקטין את הבעיה, אך מגדיל שיברור פנימי.

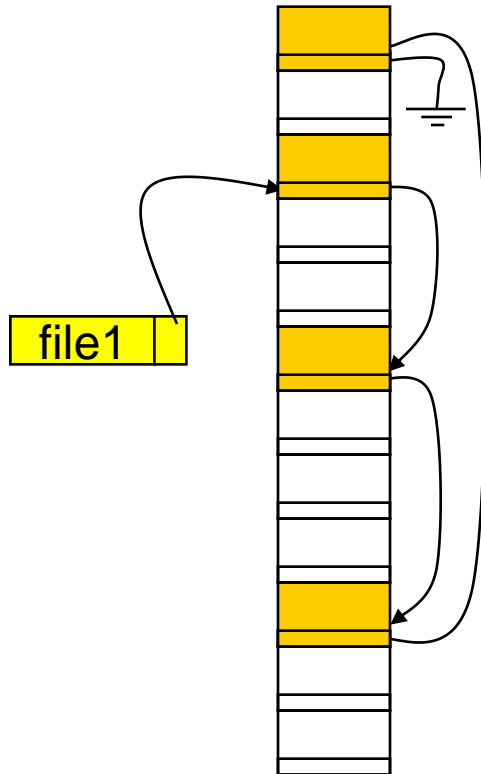
✗ שיבוש מצביע בבלוק גורם לאיבוד חלקים של קובץ.

הקצאה משורשרת: מצב הקדיסק

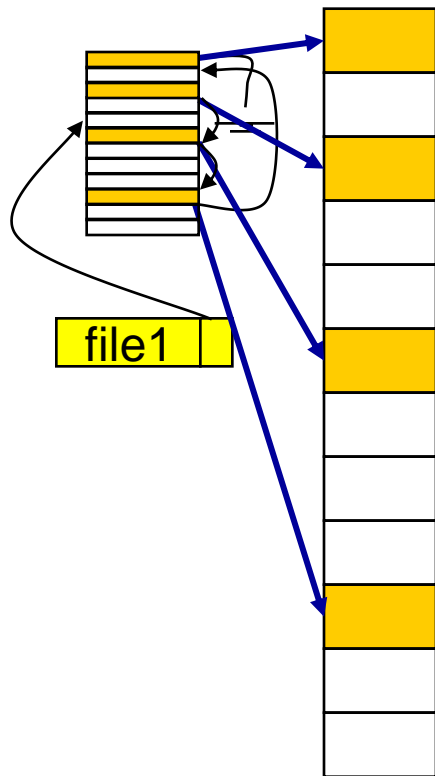


File Allocation Table (FAT)

□ החזקת שרשרת המצביעים בנפרד.



File Allocation Table (FAT)



□ החזקת שרשרת המצביעים בנפרד.

□ מצביע הקובץ (במדריך) מראה על הכניסה הראשונה.

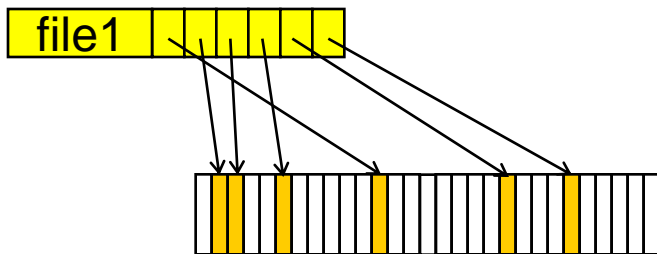
□ בעצם, טבלה שמתארת את התוכן של הדיסק כולו.

□ .MS-DOS

FAT *מכילות*

- הטבלה נמצאת במקום נוח בדיסק / זיכרון ראשי, ומכילה 2^{16} כניסות (אינדקס של 16 ביטים)
 - כאשר הדיסקים גדלים, גודל החתיכות גדל.
 - דיסק של 6.4GB היה מחולק לחתיכות של 100KB
 - ⇐ מגדיל את השיברור הפנימי (בזבוז של 10-20% הוא שכיח).
 - לדיסקים מעל 2GB, משתמשים ב-FAT-32, עם 2^{32} כניסות
 - כל קובץ דורש לפחות חתיכה אחת.
 - ⇐ לכל היותר 64K קבצים ב-FAT, 4G ב-FAT-32.
 - טבלת ה-FAT מהווה משאב קריטי.
 - צוואר בקבוק בגישה.
 - ⇐ לאובדן הטבלה או לפגיעה בה יש משמעות קטסטרופאלית
- ...ולכן היא מוחזקת בשני עותקים

מיפוי קבצים: אינדקס



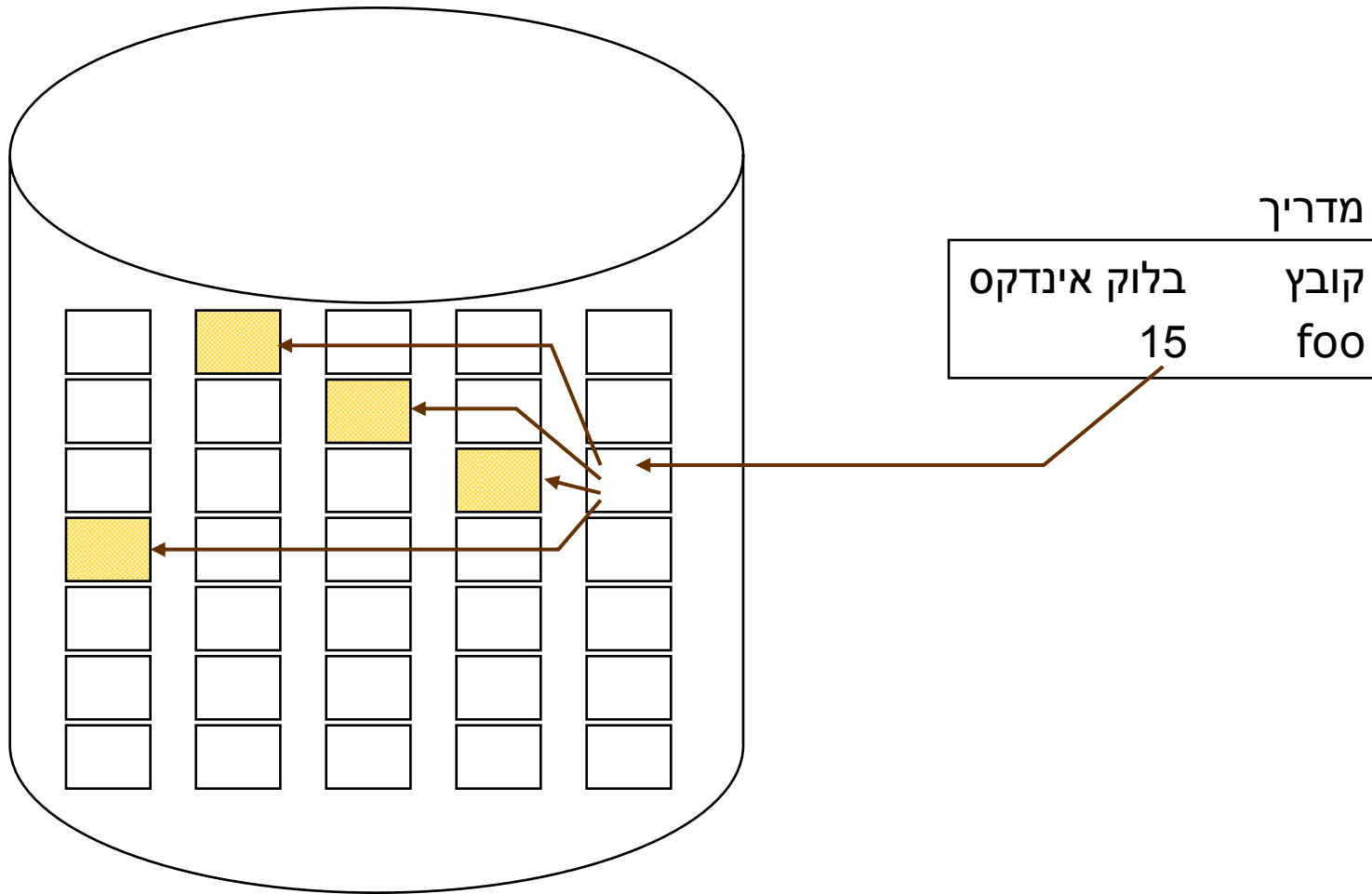
□ המשתמש מזהיר על גודל קובץ מקסימלי.

□ המצביעים לבלוקים מרוכזים בשטח רצוף

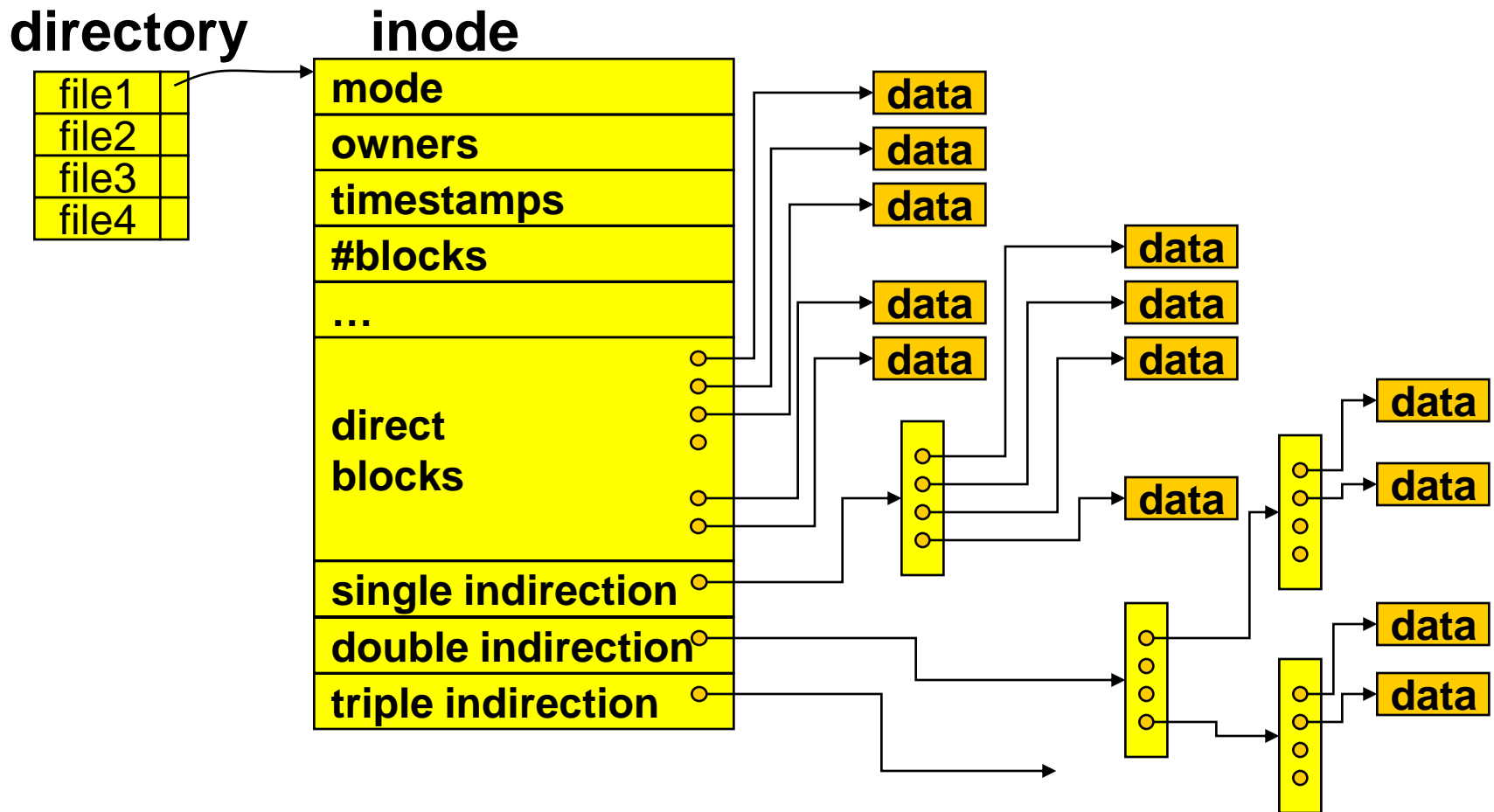
✓ משפר במקצת את זמן הגישה הישירה לעומת הקצאה משורשרת.

✗ צריך להזהיר מראש על גודל קובץ

אינדקס: מצב הדיוסק



מיפוי קבצים: אינדקס מראה כמות



מבנה מערכת הקבצים ב Unix 4.1

□ inodes (index nodes) אינם מדריכים,
אלא בלוקים רגילים

□ מדריכים הינם קבצים רגילים אשר ממפים שמות קבצים
ל inodes.

לאחר שימוש ממושך במערכת הקבצים:

□ בלוקים שונים של אותו קובץ נמצאים רחוק זה מזה.

□ inodes ובלוקי אינדקס נמצאים רחוק מבלוקים של מידע.

Unix 4.2 Fast File System (FFS)

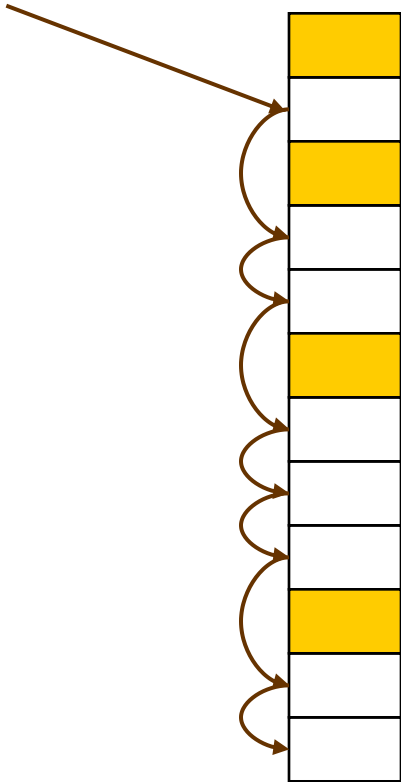
- קבוצות של צילינדרים קרובים (על הדיסק).
- בתוך אותה קבוצת צילינדרים משתדלים לשים:
 - בלוקים של אותו קובץ
 - בלוקים של אותו מדריך (כולל inodes).
- בתוך קבוצות שונות שמים:
 - בלוקים של קבצים ממדריכים שונים.
- שומרים על $\sim 10\%$ מקום פנוי בכל קבוצת צילינדרים.

ניהול בלוקים פנויים: bitmap

0	
1	
0	
1	
1	
0	
1	
1	
1	
0	
1	
1	

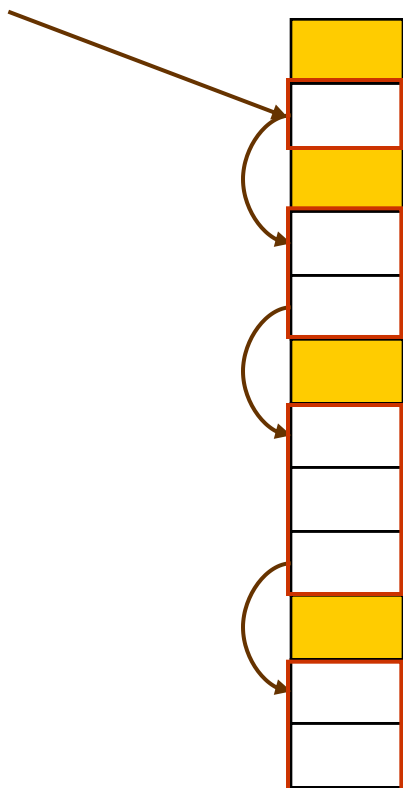
- מערך ובו סיבית לכל בלוק.
- 0 – הבלוק תפוס; 1 הבלוק פנוי.
- מאוחסן במקום קבוע בדיסק.
- עותק בזיכרון הראשי, ליעילות.
- עם בלוקים של 4KB, נזדקק לבלוק של ביטים עבור 8×4096 בלוקים.
- קל לזהות רצף של בלוקים פנויים.

ניהול בלוקים פנויים: רשימה מקושרת



- ❑ מציאת בלוק פנוי בודד מהירה.
- ❑ הקצאה של מספר בלוקים לאותו קובץ:
 - במקומות המרוחקים זה מזה.
 - מציאת הבלוקים מחייבת תזוזות של הדיסק.
- ❑ מבנה הנתונים נהרס אם בלוק באמצע הרשימה השתבש.
- ❑ ב FAT, הבלוקים הפנויים מנוהלים כקובץ אחד.

ניהול בלוקים פנויים: קיבול



□ שימוש ברשימה מקושרת של אלמנטים:

■ grouping - כל אלמנט מכיל טבלה של מצביעים לבלוקים פנויים רצופים ומצביע לאלמנט הבא

■ counting - כל אלמנט מכיל מצביע לבלוק הפנוי הראשון מקבוצת בלוקים פנויים רצופים, מספר הבלוקים בקבוצה ומצביע לאלמנט הבא

□ ניתן למצוא בצורה יעילה מספר גדול של בלוקים פנויים ורציפים.

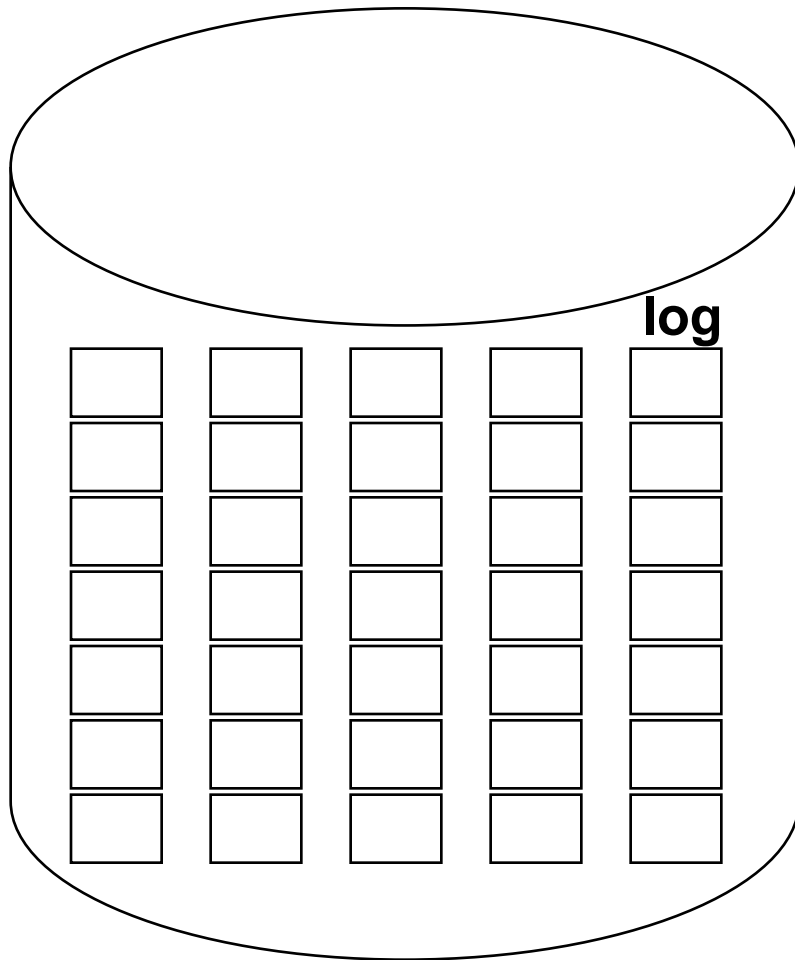
אמנות

- המידע בדיסק מתחלק ל-
 - user data: נתוני המשתמש (בתוך הקבצים).
 - metadata: מידע על ארגון הקבצים.
 - בלוקים של אינדקס, inodes...
- איבוד/שיבוש metadata עלול לגרום לאיבוד user data רב.
 - נפילת חשמל באמצע כתיבה עלולה לשבש את הסקטור שכעת נכתב.
- מתי כתיבות עוברות מהזיכרון הראשי לדיסק?
 - write-through – כל כתיבה עוברת מיידית לדיסק.
 - write-back – הדיסק מעודכן באופן אסינכרוני, אולי לא לפי סדר.

אמ'נות ק-Unix

- ה data נכתב במדיניות write-back: נתוני המשתמש נכתבים באופן מחזורי לדיסק.
- פקודות sync, fsync מאלצות לכתוב את הבלוקים המלוכלכים לדיסק.
- ה metadata נכתב במדיניות write-through: עדכונים נכתבים מידית לדיסק
- נתונים מסוימים נשמרים במספר עותקים
- למשל, שורש ה-file system
- כאשר מערכת הקבצים עולה אחרי נפילה, מתקנים את מבני הנתונים. במערכות קבצים מסוימות, דורש מעבר על כל הדיסק.
- ScanDisk, fsck (file system check)

רישום (logging)



- שיטה יעילה לתחזוקת ה-metadata
- רושמים ב-log סדרתי את העדכונים לפני שהם נכתבים לדיסק (write-ahead logging).
- הבלוקים שהשתנו נכתבים לדיסק לאחר-מכן.
 - אולי לא לפי הסדר.
 - אפשר לקבץ מספר שינויים ולכתוב אותם בכתובה אחת.

התאווט עט log

□ לאחר נפילה, עוברים על כל הכניסות ב-log

■ בצע את הפעולה.

□ ביצוע נוסף של פקודה שהתבצעה במלואה או חלקית נותן תוצאה שקולה לביצוע הפעולה המקורית (idempotent).

■ אם הכניסה האחרונה ב-log אינה שלמה, מתעלמים ממנה.

□ ניתן למחוק מה-log עדכונים שכבר נכתבו לדיסק.

יתרונות ומסכנות *fe* logging

✓ כתיבה לדיסק באופן אסינכרוני

✓ התאוששות יעילה.

■ תלויה במספר השינויים שלא נכתבו לדיסק, ולא בגודל מערכת הקבצים.

✗ דורש כתיבות נוספות.

■ בערך מכפיל את מספר הכתיבות.

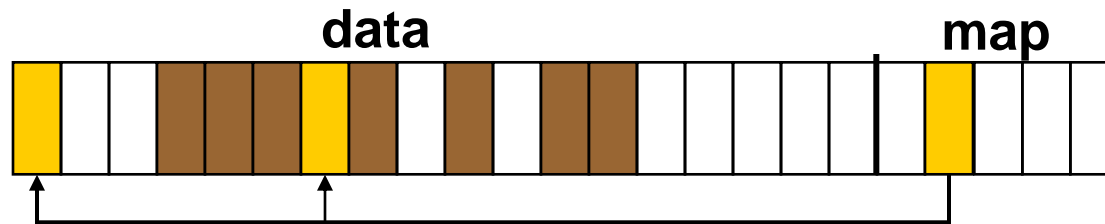
מערכת קבצים log-structured

- ה log הוא העותק היחיד של הנתונים
- כאשר משנים בלוק (header, data) פשוט כותבים את הבלוק החדש ל log.
- בסוף ה-log, ולכן העותק הישן של הבלוק לא תקף
- מפה מיוחדת מאפשרת לדעת איפה כל בלוק נמצא ב log

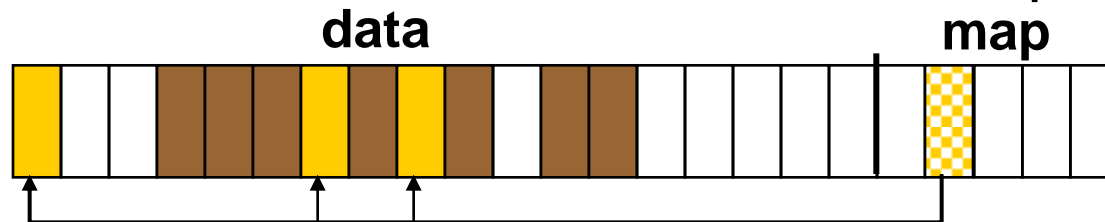
המקור: דיסקים אופטיים שניתן לכתוב רק פעם אחת (write-once)

במערכת קבצים "מסורתי"

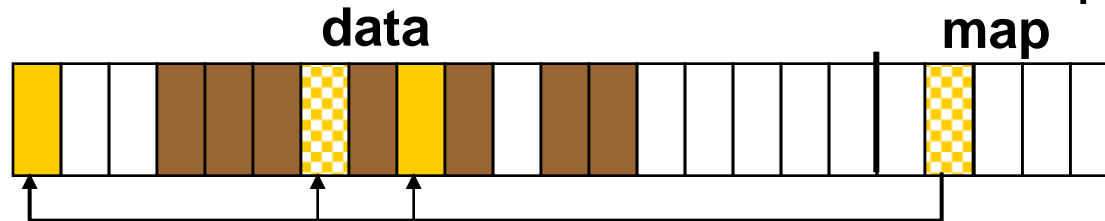
לפני:



אחרי הוספת בלוק:

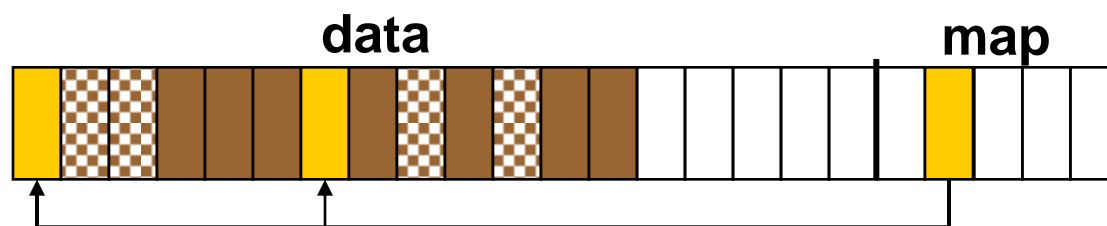


אחרי שינוי בלוק:

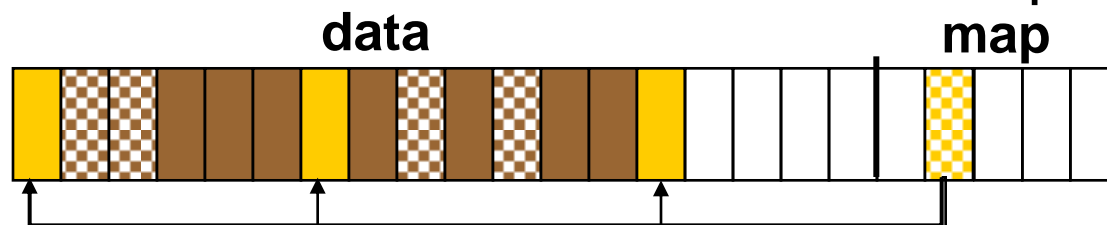


Log במערכת מבוססת

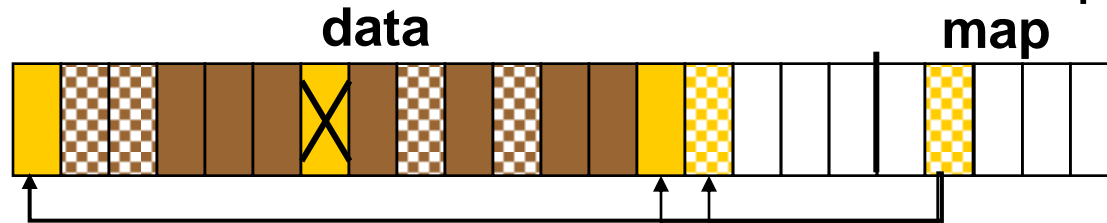
לפני:



אחרי הוספת בלוק:



אחרי שינוי בלוק:



LFS: יתרונות וחסרונות

✓ הכתיבות הן סדרתיות (ולכן יעילות יותר)

✓ יש seeks רק עבור קריאות

✓ עם מטמון גדול בזיכרון הראשי, רוב הקריאות לא הולכות לדיסק

✓ בכתיבה סדרתית של קובץ, סיכוי טוב שהבלוקים של הקובץ רצופים בדיסק (ב \log), מבטיח קריאה יעילה בעתיד.

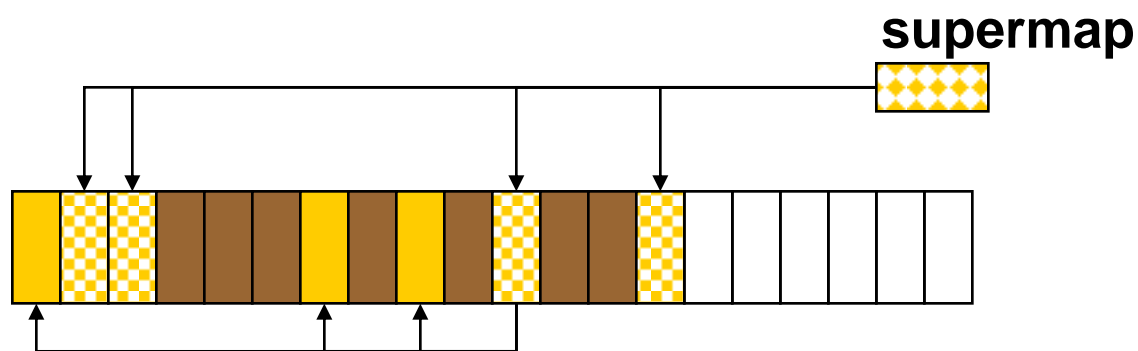
✗ איסוף בלוקים שמתפנים (מה קורה אם הדיסק מתמלא?)

✗ הזזת הבלוקים התפוסים לתחילת ה- \log

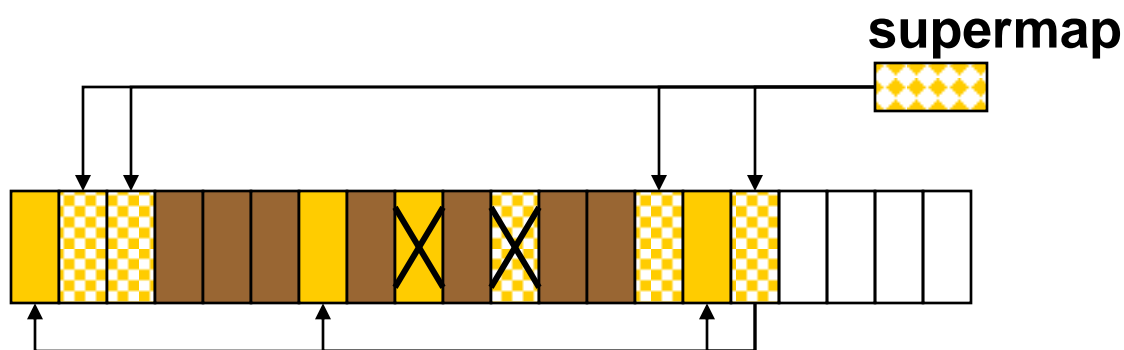
למה את המפה כותבים ב-Log

supermap מאפשר למצוא את הבלוקים המעודכנים של המפה

לפני:



אחרי הוספת בלוק (שינוי בלוק מבוצע באופן דומה):



LFS *מימוש*

המימוש הראשון ב-1992 עבור מערכת ההפעלה Sprite
על-ידי Ousterhout and Rosenblum

BSD-LFS מימוש ל BSD (היוניקס של ברקלי),
נמצא ב NetBSD

LogFS הוא מימוש ל Linux עבור זיכרון פלאש.

Free-Space Management

כיצד נמצא מקומות פנויים בדיסק:

1. מערך ביטים – נחזיק ווקטור בלוקים פנויים מ-0 עד $n-1$. כל בלוק מיוצג ע"י ביט אחד בלבד. אם הבלוק תפוס יופיע הביט 1, אחרת 0. כאשר רוצים להגדיל קובץ, מחפשים בלוק שהביט שלו 0 ומוסיפים לקובץ שלי.

חסרון: בזבוז – מתקבל ווקטור ענק שתמיד נשמר בזיכרון, גם כאשר אין אף בלוק פנוי.

1. רשימה מקושרת – רשימה מקושרת של בלוקים פנויים. בלוק שמתפנה מתווסף לרשימה ולהפך. אפשרות יעילה יותר היא שכל בלוק פנוי מצביע לבלוק הפנוי הבא מיד אחריו, ואז אם צריכים X בלוקים פנויים ניגשים לאחר ומוצאים את היתר. שיטה זו לא תתאים כאשר עובדים על cluster-ים.

חסרון: לא יעיל – איטי.

יתרון: חסכון במקום.

1. Grouping – בלוק ראשון מצביע על n בלוקים פנויים ובנוסף על בלוק האינדקסים הבא (הבנוי באותה צורה). מחזיקים את הבלוקים בקבוצות, אבל אין קשר למיקום הפיזי.

2. Counting – מחזיקים מצביע לבלוק פנוי (כתובת) וכמה בלוקים פנויים יש אחריו. ההתייחסות כאן היא פיזית איך הבלוקים יושבים. זוהי השיטה היעילה ביותר מבחינת מהירות.

מערכת הקבצים מחולקת לוגית למספר שכבות:

תוכניות יישום (application programs)



מערכת הקבצים הלוגית (LFS—Logical file system)



מערכת ההפעלה משתמש ב-device-ים לצורך טיפול בבקשות



מודל ארגון הקבצים (FOM—file organization module)



מערכת הקבצים הבסיסית (BFS—basic file system)



I/O control



התקנים (devices)

התקנים (devices)

כל רמה משתמשת ב-feature-ים של רמות נמוכות יותר ומייצרת feature-ים חדשים לרמות הגבוהות.

- I/O control – מורכבת מ-device drivers

- ו-interrupt handler להעברת מידע בין

- הזיכרון למערכת הדיסקים. Device driver

- יכול להיחשב כמתרגם. הקלט שלו מורכב מפקודות

- בשפה גבוהה, ואילו הפלט שלו נתון ברמה נמוכה – הוראות חומרה

- מדויקת, בהם משתמש הבקר החומרה.

- Basic file system – מוציא פקודות להתקנים לקרוא ולכתוב בלוקים (רמה פיזית).

- File-organization module – מכיר גם רמה לוגית ויכול לתרגמה לרמה הפיזית עבור ה-BFS.

- נעזר גם ב-free space manager כדי למצוא מקומות ריקים בדיסק.

- Logical file system – משתמש ב-directory structure כדי לספק ל-FOM את המידע לו הוא

- זקוק, בהינתן שם קובץ. אחרי גם להגן על הקבצים.

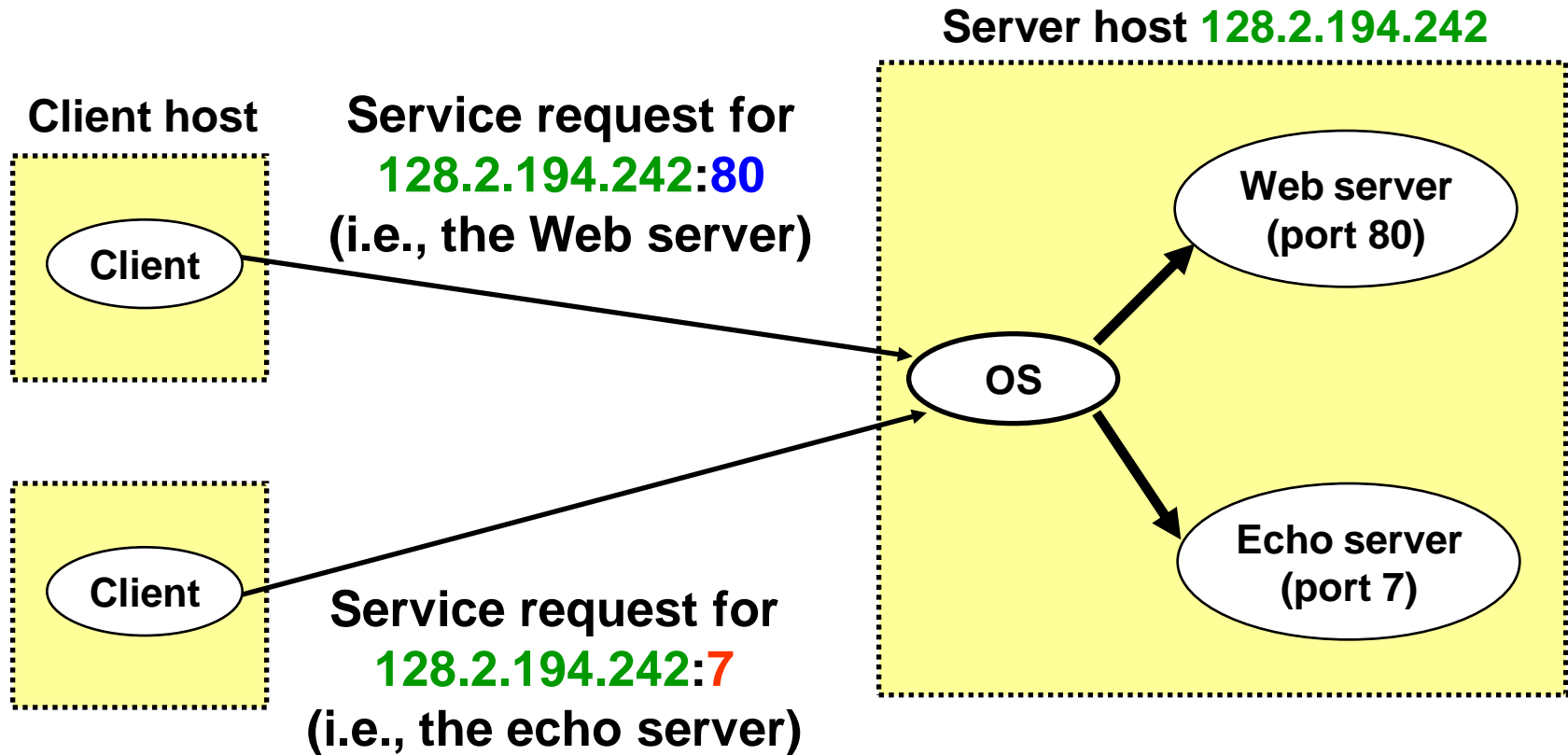
אפליקציות שרת-לקוח

- פרדיגמת שרת לקוח
- מושג ה socket
- מבנה שרת-לקוח
- קצת יותר על רשתות

זיהוי התהליך המקבל

- התהליך השולח צריך לזהות את התהליך המקבל
 - שם או כתובת של מחשב הקצה
 - מזהה של התהליך המקבל
- זיהוי מחשב הקצה באופן ייחודי
 - על-ידי כתובת IP (Internet Protocol) בת 32 ביטים
- זיהוי התהליך המקבל
 - מחשב הקצה מריץ הרבה תהליכים
 - זיהוי התהליך על-ידי **port number** בן 16 ביטים
 - מושג של רשת התקשורת ולא של מערכת ההפעלה

זיהוי תהליך אף-ידי כורט



באיזה פורט להשתמש?

□ לאפליקציות פופולאריות יש מספר פורט ידוע

■ למשל 80 לשרת web, 25 לשרת mail

■ רשימה ב <http://www.iana.org>

□ פורטים ידועים ופורטים זמניים

■ לשרת יש מספר פורט ידוע (למשל 80)

□ בין 0 ל- 1023

■ הקליינט מתקשר דרך מספר פורט זמני

□ בין 1024 ל 65535

מבנה טיפוסים לקוח

□ הכנה להתקשרות:

■ יצירת socket

`socket(...)`

■ מציאת כתובת השרת ומספר הפורט (בד"כ קבוע)

■ אתחול התקשורת לשרת

`connect(sockfd, serv_addr, ...)`

□ החלפת מידע עם השרת

■ כתוב מידע ל socket

■ קרא מידע מה socket

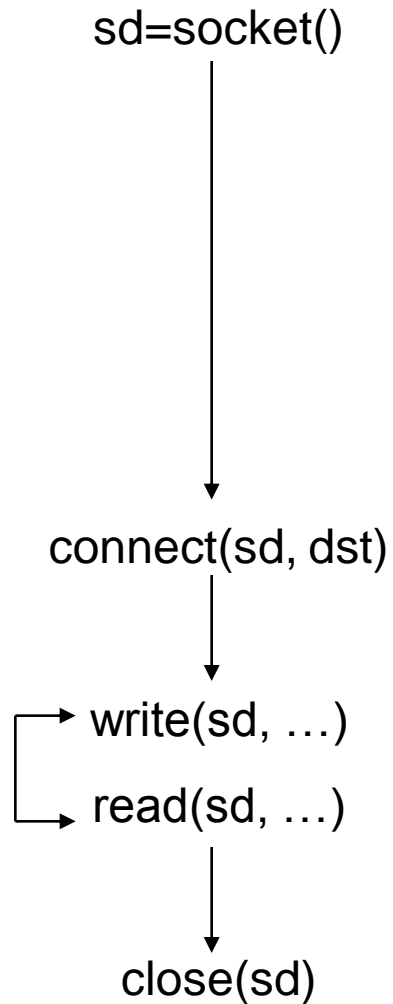
■ טפל במידע (למשל, הצג דף html)

□ סגור את ה socket

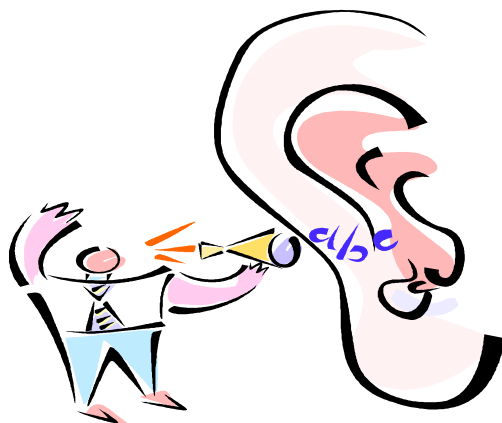
`close(sockfd)`

control flow : חיפוש-הנהגה

Client side



שרתים אינם לקוחות



□ פאסיביים: מוכנים לתקשורת

■ אבל לא מתחילים עד ששומעים מהלקוח

□ שומעים מכמה לקוחות

■ צריך לאפשר תור של לקוחות מחכים,

אם מספר לקוחות מתחילים התקשרות בו-זמנית

□ יצירת socket לכל לקוח

■ כאשר מקבלים בקשה מלקוח חדש,

יוצרים עבורו socket חדש (ובלעדי)

מבנה טיפוסים של שרת

□ הכנה להתקשרות:

■ יצירת socket

`socket(...)`

■ קשר כתובת עצמית ומספר פורט עם ה-socket

`bind(sockfd, my_addr, ...)`

מבנה טיפוס *fd* שרת: הקשמה

□ המתן לשמוע מלקוח (passive open)

■ ציין כמה לקוחות ממתנים מותרים

```
listen(sockfd, num)
```

□ קבל בקשת התקשרות מלקוח ויצור socket יעודי

```
accept(sockfd, &addr, ... )
```

□ החלפת מידע עם הלקוח על ה-socket החדש

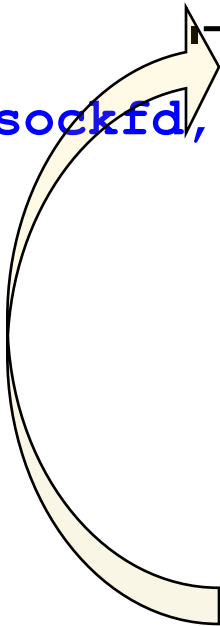
■ קרא מידע מה socket

■ טפל בבקשה (למשל, הבא קובץ html)

■ כתוב מידע ל socket

■ סגור את ה-socket

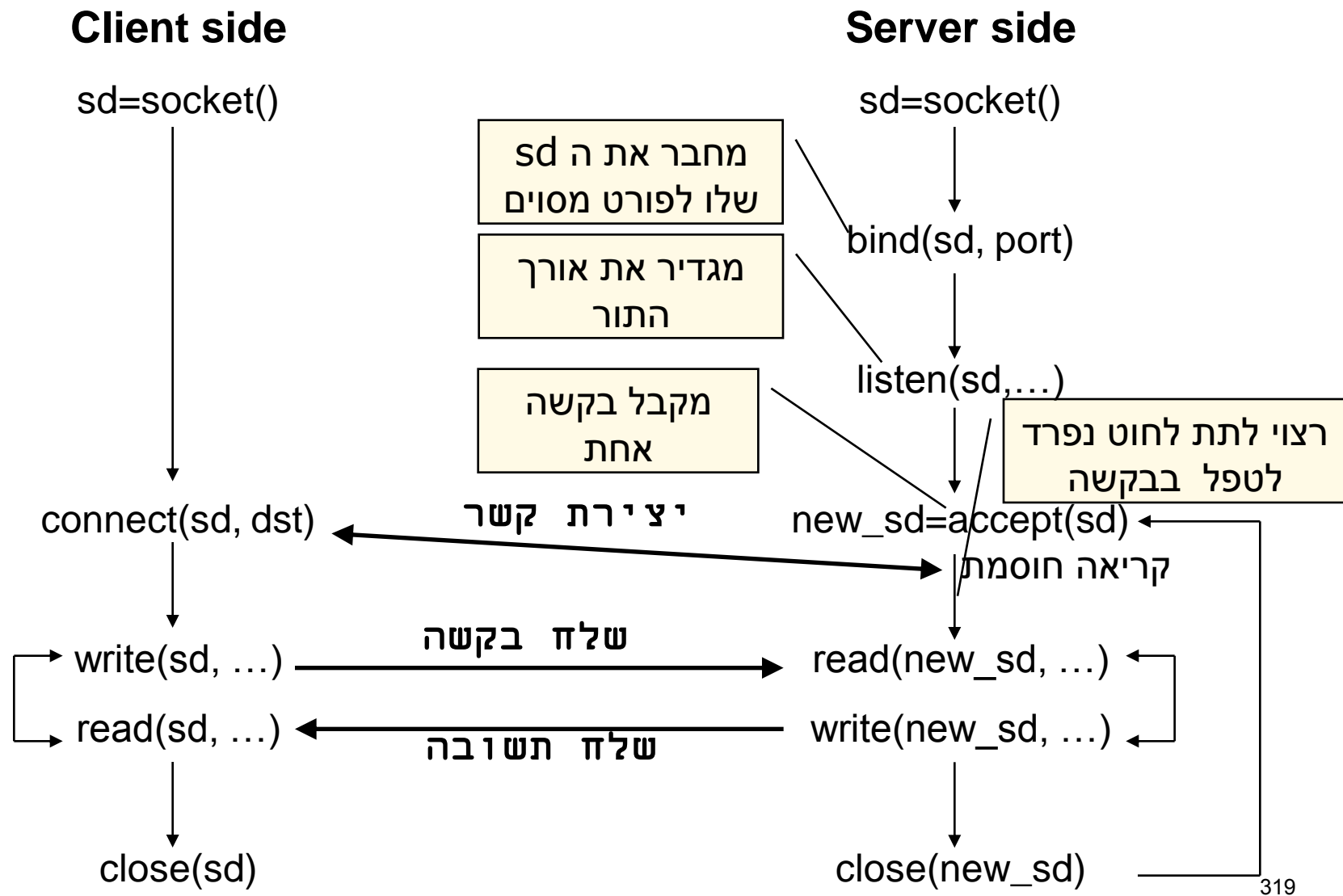
□ חזור להמתנה



טיפול בבקשות

- טיפול סדרתי בבקשות אינו יעיל
 - כל שאר הקליינטים צריכים להמתין
- כדאי שהשרת יטפל בכמה בקשות קליינטים בו-זמנית (time-sharing)
 - קצת עבודה על בקשה אחת, ועבור לבקשה אחרת
 - פירוק למשימות קטנות, כמו מציאת קובץ...
 - או יצירת תהליך חדש לטיפול בכל בקשה
 - או שימוש במאגר חוטים (כמו שראינו בהתחלת הקורס)

מימוש פת-פקות: control flow



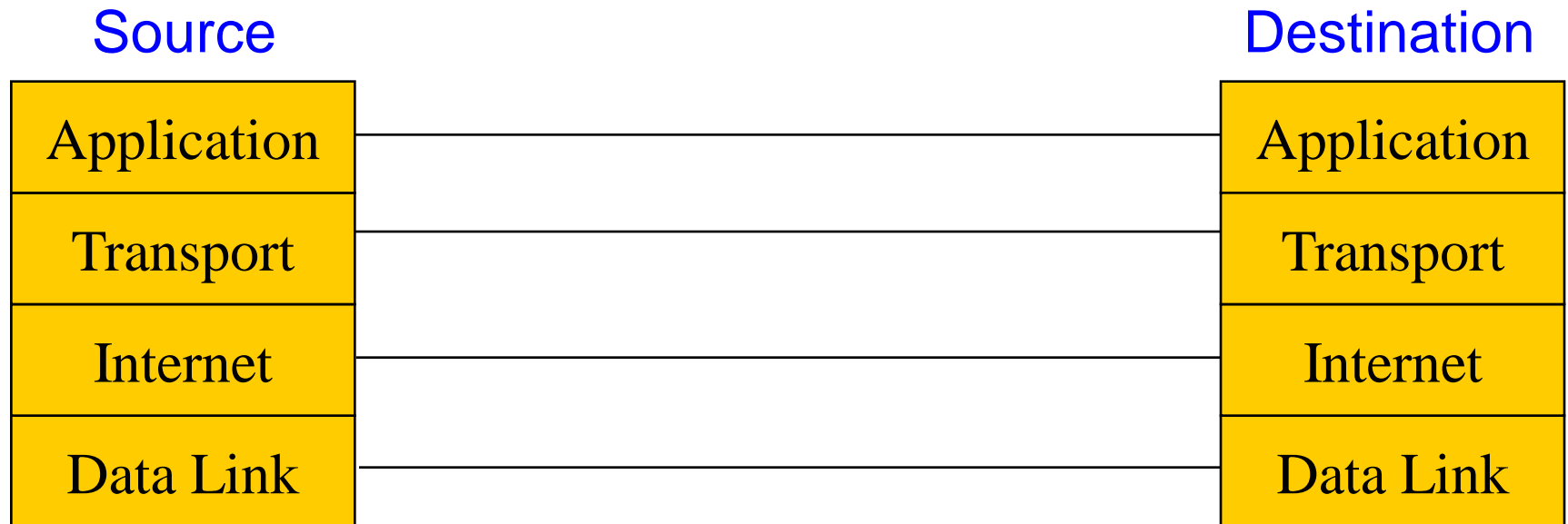
מתחת ל socket :מודל השכבות

- פרוטוקולי תקשורת מאפשרים העברת נתונים ברשת, ופועלים בשכבות שונות, כשלכל שכבה תפקיד משלה.
 - אוסף פרוטוקולי התקשורת הנפוץ ביותר נקרא TCP/IP
- מכיל 4 שכבות

Application (telnet, ftp ...)	אפליקציות המשתמשות ברשת
Transport (TCP, UDP)	תקשורת בין תהליכים (ולא מחשבים)
Internet (IP)	ניתוב חבילות בין תחנות (לא שכבות)
Data Link	העברת חבילה בין תחנות שכנות

תקשורת בשכבות

כל שכבה מקבלת שירותים מהשכבה שמתחתיה
מספקת שירותים לשכבה שמעליה
ו"מדברת" עם השכבה המקבילה במחשב השני



שכבת Transport: TCP / UDP

- מאפשרת תקשורת בין תהליכים (שכבת IP מאפשרת תקשורת בין מחשבים)
- UDP (User Datagram Protocol):
מעביר הודעה בודדת בין שני תהליכים,
מבלי להבטיח סדר או אמינות
- TCP (Transport Connection Protocol)
יוצר session בין שני תהליכים, ומבטיח:
 - סדר: החבילות יגיעו ליעדן בסדר שבו נשלחו
 - אמינות: כל החבילות יגיעו ליעדן (חבילה שהולכת לאיבוד משודרת מחדש)

מערכות קבצים מבוזרות

□ מבוא

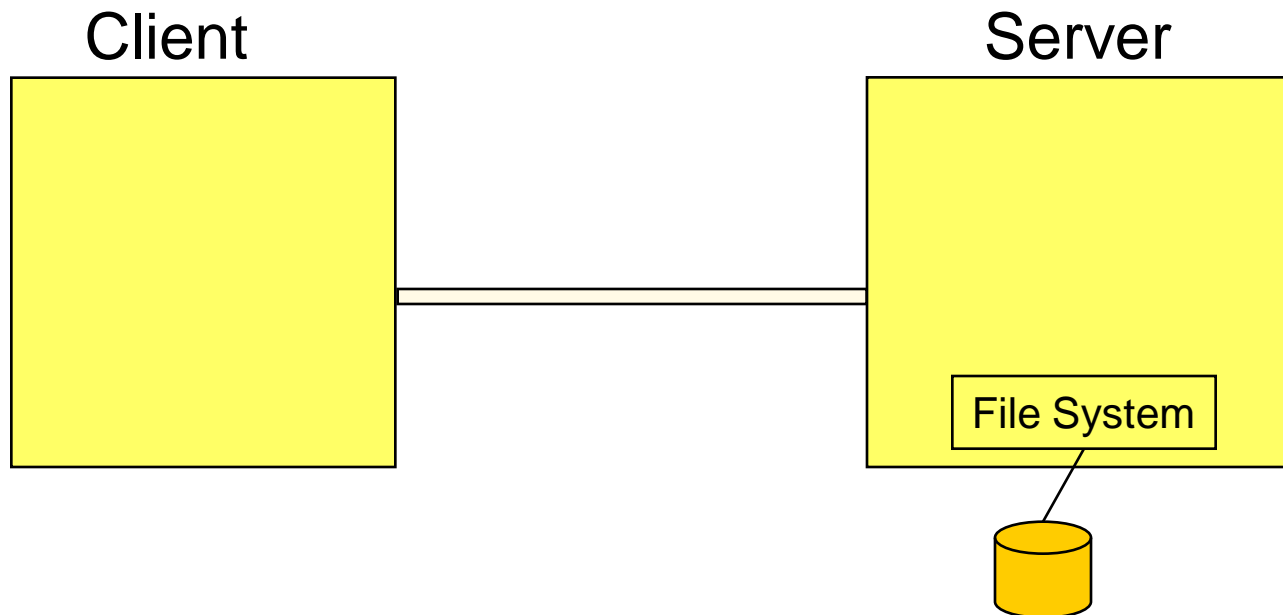
□ מבנה כללי

□ דוגמה: Network file system

□ דוגמה: Google file system

התמונה הסדולה

- תהליך רץ במחשב **לקוח** (client)
- התהליך מבקש לגשת לקובץ הנמצא במחשב **שרת** (server)
- מחשב הלקוח ומחשב השרת מחוברים באמצעי תקשורת כלשהו (כבל תקשורת, רשת מקומית, אינטרנט)

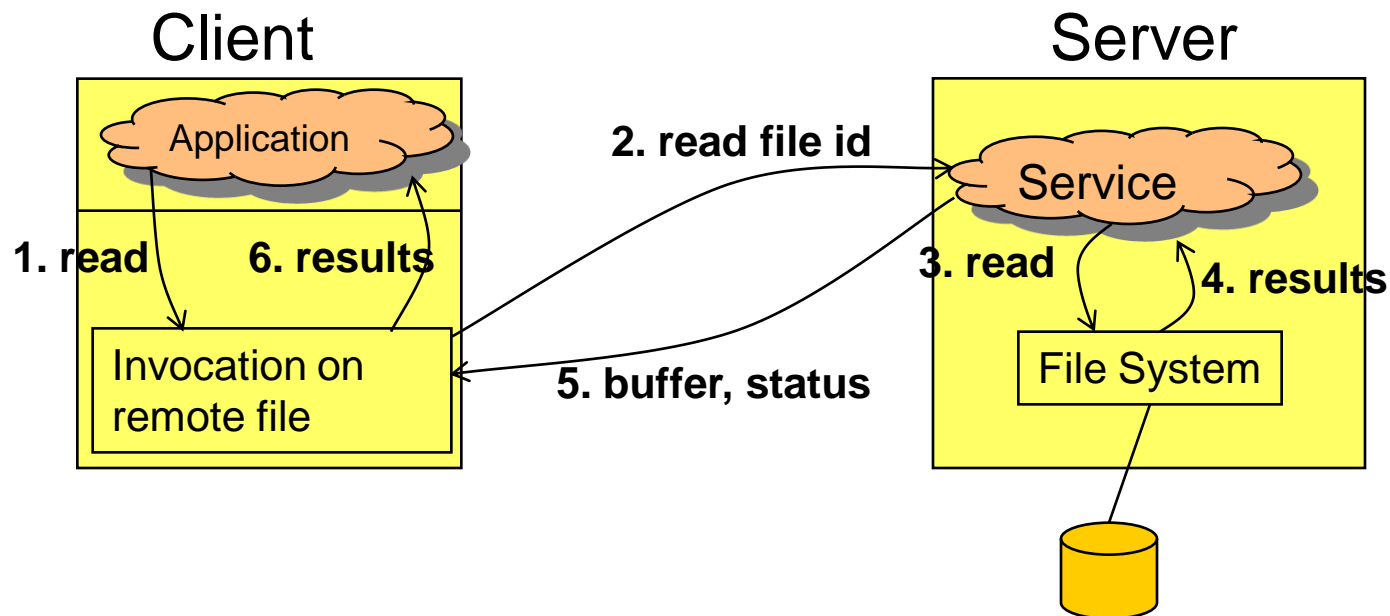


התמונה האדומה

- תהליך רץ במחשב **לקוח** (client)
- התהליך מבקש לגשת לקובץ הנמצא במחשב **שרת** (server)
- מחשב הלקוח ומחשב השרת מחוברים באמצעי תקשורת כלשהו (כבל תקשורת, רשת מקומית, אינטרנט)
- כאשר תהליכים בלקוח מבצעים קריאות מערכת לקבצים אצל השרת:
 - הלקוח שולח לשרת **בקשות**
 - מערכת ההפעלה שולחת בקשות בהתאם לקריאות המערכת של התהליכים
 - השרת מחזיר **תשובות**
 - תהליך מיוחד בשרת מאזין לבקשות, מבצע אותן, ומחזיר תשובות.
 - סוגי ההודעות בין הלקוח והשרת מוגדרים ב**פרוטוקול תקשורת**

אפשרות 1: שיטת פונקציה

הפעולות מועברות לשרת אשר מבצע אותן לוקלית ומעביר את התוצאות חזרה ללקוח (Function Shipping)
✓ מימוש פשוט (למשל Remote Procedure Call)

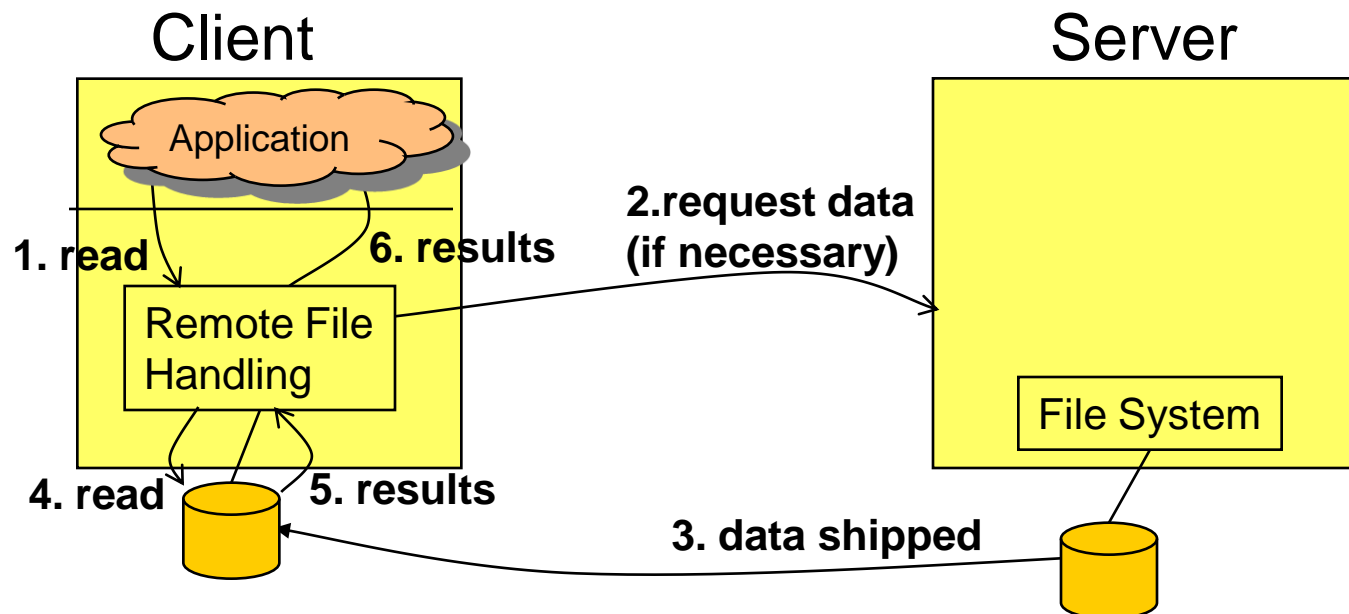


אפשרות 2: שיאור נתונים

הנתונים מועברים מהשרת ללקוח, אשר מבצע את הפעולה באופן מקומי (Data Shipping)

✓ מוריד עומס בשרת

✗ איך יודעים האם הנתונים עדכניים?



מדיניות משובצת: מטמון

□ מדיניות עדכון: מתי מעבירים עדכונים מהמטמון לשרת?

■ write-through: אמין, אך המטמון מנוצל אך ורק בקריאה

■ write-back: עדכונים אחרונים עלולים ללכת לאיבוד

■ write-on-close: השינויים מועברים בעת סגירת הקובץ

■ delayed-write: השינויים מועברים כל פרק זמן

□ קונסיסטנטיות: איך הלקוח יודע שהנתונים במטמון תקפים?

■ client-initiated: הלקוח בודק בכל גישה, כל פרק זמן או כאשר הקובץ נפתח

■ server-initiated: השרת זוכר איזה לקוח מחזיק איזה חלקים של קבצים

□ השרת יכול לשלוח הודעה revoke ללקוח אשר פוסל את העותק שלו

□ או, עבור קבצים הפתוחים לכתיבה ע"י מספר לקוחות, מבטלים את המטמון וניגשים כל הזמן לשרת

יש מצב?

האם השרת שומר מצב עבור כל לקוח בין בקשה לבקשה?

■ למשל, איזה קבצים נפתחו ע"י הלקוח, מיקום בקובץ, מידע על הנתונים במטמון של הלקוח, מנעולים וכדומה.

□ פרוטוקול עם מצב (stateful).

■ השרת שומר מידע לגבי כל לקוח.

■ בקשה מתבצעת בהקשר מסוים, ואין צורך להעביר את כל המידע הדרוש לביצוע הפקודה.

אין מצב!

□ פרוטוקול בלי מצב (stateless).

■ השרת לא שומר מידע.

■ בקשה מכילות את כל המידע הנחוץ לטיפול בהן.

✓ קל יותר למימוש.

✓ קל להתאושש מפגמים.

■ השרת יכול ליפול ולהתאושש מבלי שלקוחות ירגישו (חוץ מאשר האטת זמן התגובה בעת ההתאוששות).

✗ לא ניתן לבצע שיפורים ולחסוך בתקשורת

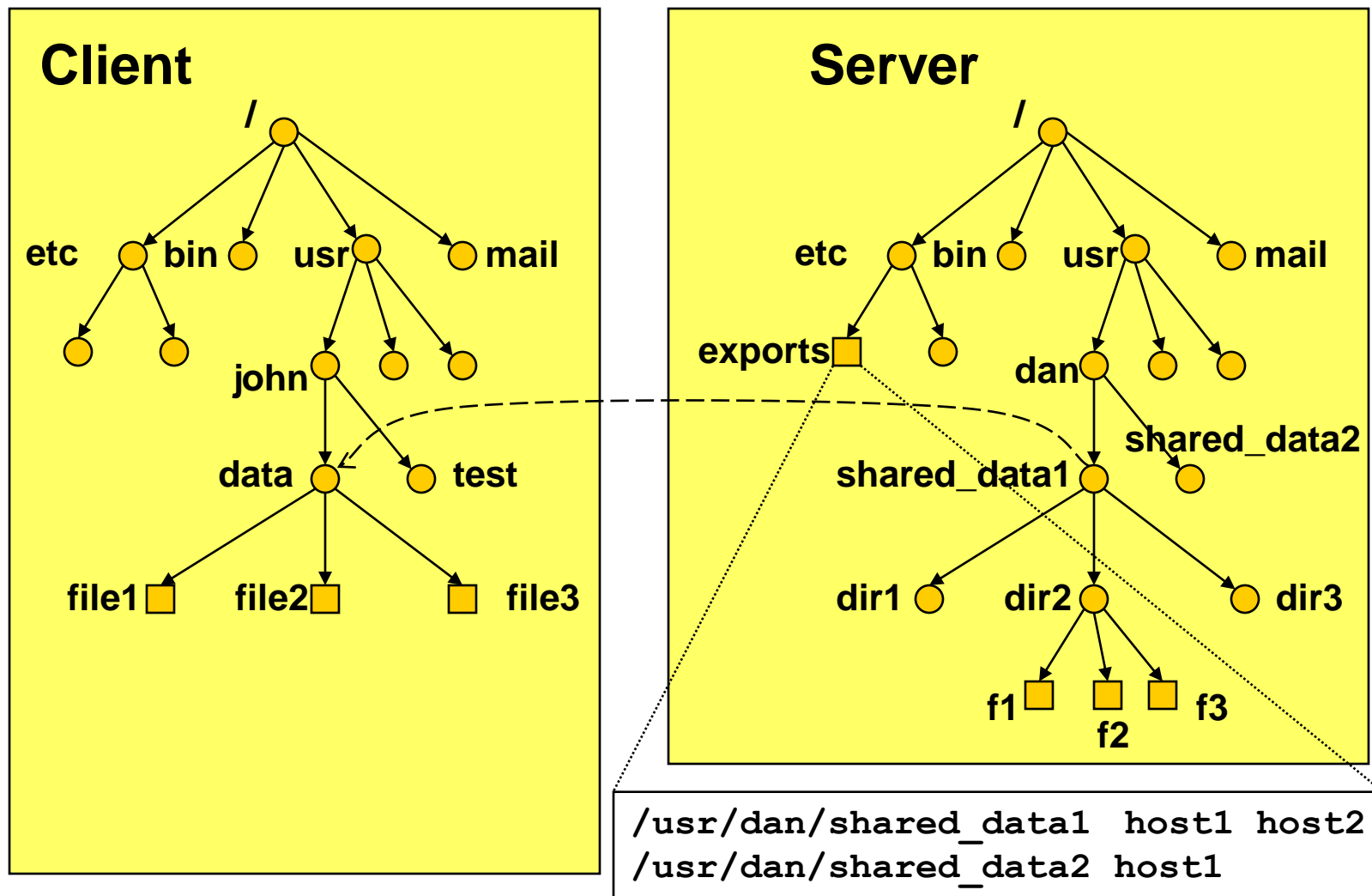
✗ קשה לממש נעילות של קבצים

■ השרת לא יכול לזכור שהקובץ נעול

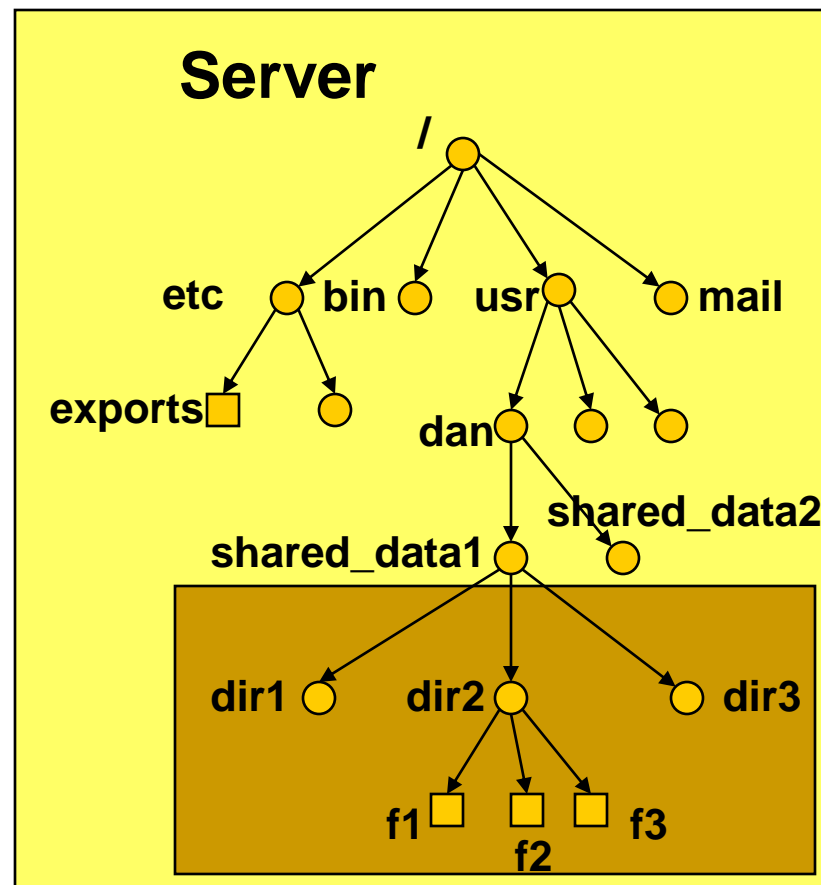
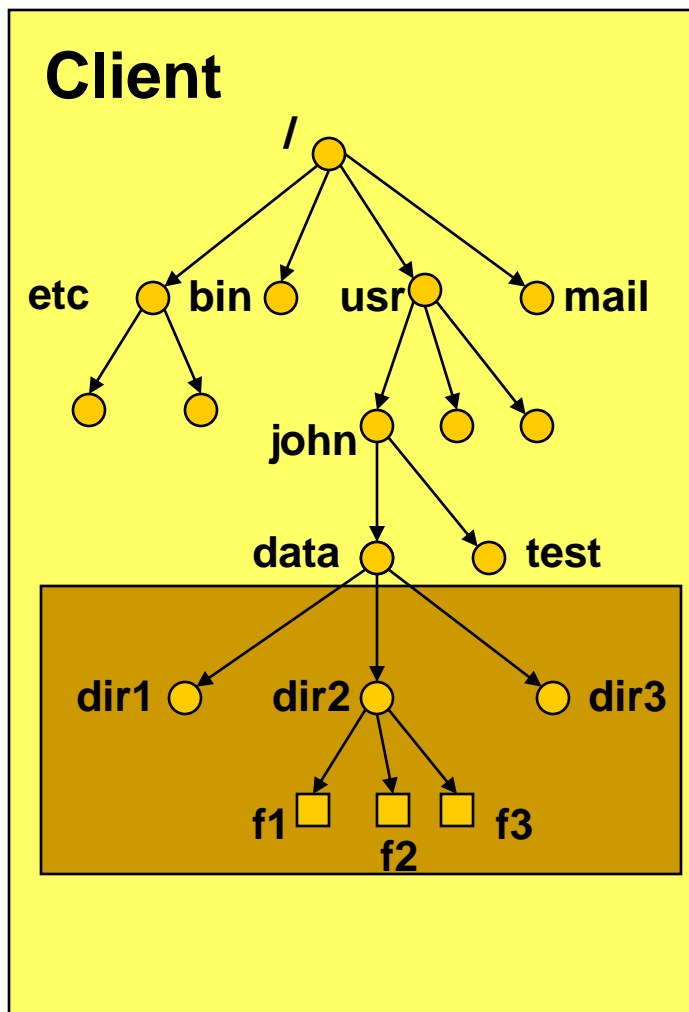
Network File System: NFS

- פרוטוקול שרת-לקוח.
- בעיקרון, בלי מצב (stateless).
- פעולות (RPC) Remote procedure call:
 - read ו-write על קובץ
 - גישה לתכונות של קובץ
 - חיפוש בתוך מדריך
 - פעולות על מדריכים, כמו חיפוש, הוספת/מחיקת כניסה וכד'
 - אין בפרוטוקול פעולות open ו-close
- הלקוח מרכיב (mounts) תת-עץ של השרת במדריך שלו.

הרכבת מדריכים: Mount



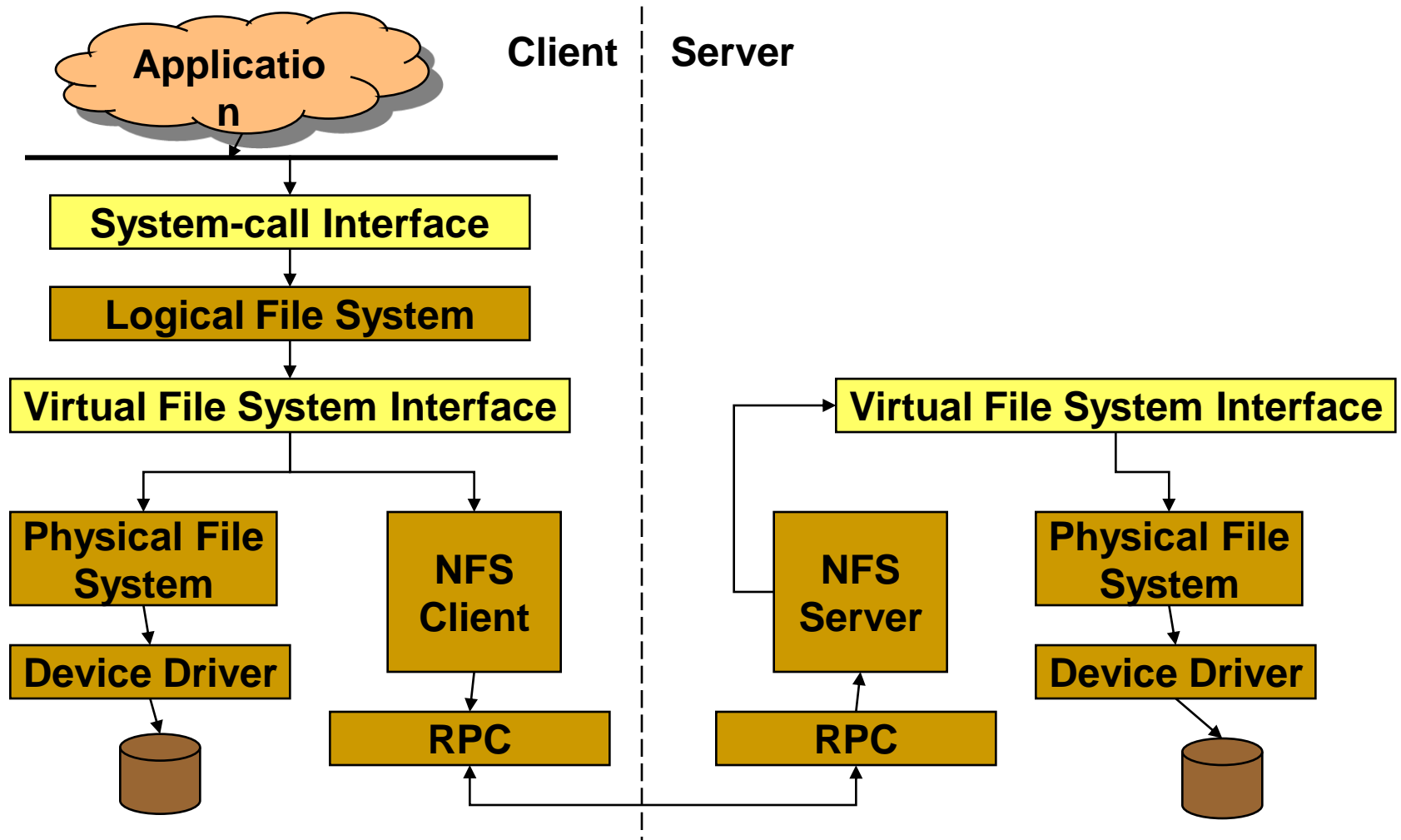
הרכבת מדריכים: Mount



שקיפות ואי-תלות ב NFS

- שקוף לאפליקציה: גישה כמו לקובץ מקומי.
- שקיפות מקום, המשתמש לא מבחין מהו מיקום הקובץ.
 - אלא אם ה-mount points ידועים לו
- יש תלות במיקום: הזזת קובץ מחייבת שינוי ה-mount

NFS e/N'N



ביצוע פקודות

- NFS מבוסס על שיגור הפונקציה לביצוע אל השרת
 - לשיפור הביצועים, משתמשים במטמון (cache) אצל הלקוח שאליו קוראים חלקים מהקובץ שאיתו עובדים
 - עדכונים לקובץ נאגרים במטמון ונשלחים לשרת מדי פרק זמן
 - כתיבות לקובץ אינן משפיעות מיד אצל לקוחות אחרים!
- כמעט כל קריאת מערכת (לגישה לקובץ) מתורגמת ישירות לפעולת RPC.
- היוצאים-מן-הכלל הם open ו-close, אשר מחייבות פעולות ניהול מקומיות בלקוח.

*Google *fe* מערכת הקבצים

□ מערכת קבצים ייעודית לתמיכה באפליקציות ספציפיות

■ אוספים מידע מכל הרשת (crawling)

■ מאחסנים על "דיסק אחד גדול"

■ מבצעים חיפוש של לקוחות על "PC אחד גדול"

■ אבל דורש הרבה יותר זיכרון וכוח חישוב ממה שמחשב בודד יכול לתת

■ בלי להשתמש במחשב מקבילי עצום



* לפי המאמר:

Ghemawat, S., Gobioff, H., and Leung, S.
The Google file system. SOSP 2003

שימוש במצבור של שרתים לולאים

- המון (המון) שרתים, כל אחד עם דיסק ו CPU.
- מאות ואלפים של יחידות PC סטנדרטיות לחלוטין וזולות
- איך לפזר מידע על פני השרתים?
- איך לטפל בנפילות?
- נפילות של דיסקים, שגיאות תכנות, בעיות חשמל, טעויות אנוש, בעיות תקשורת, ועוד ועוד

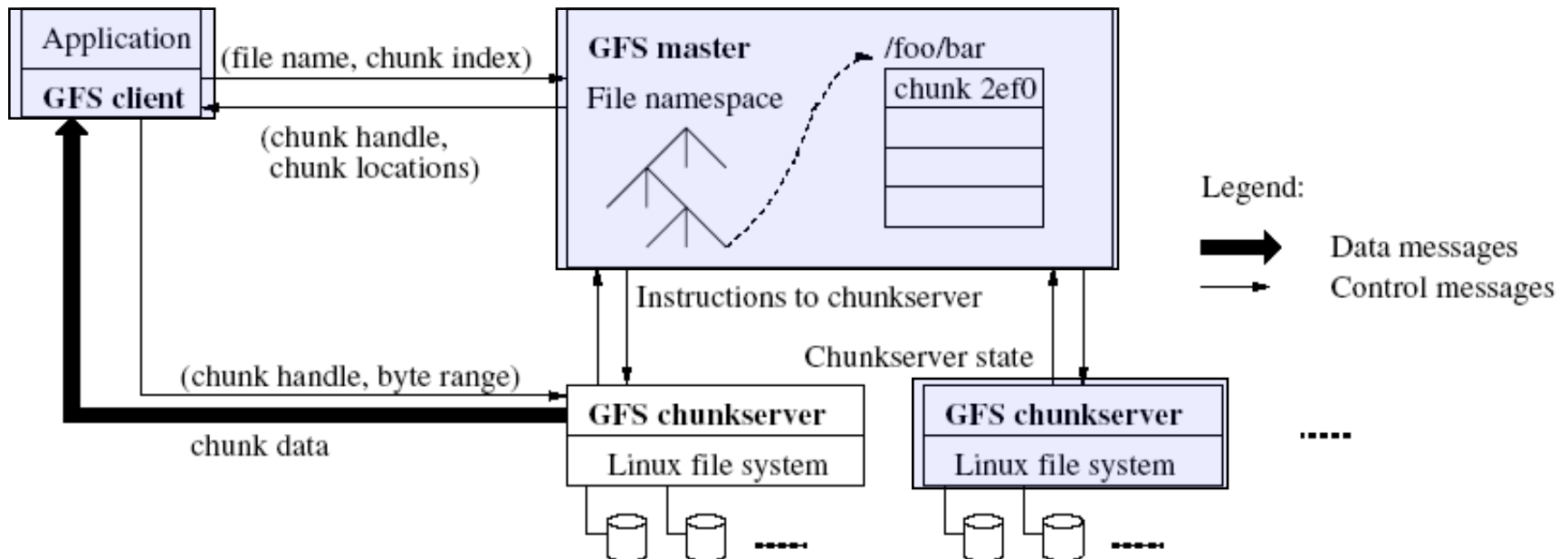
GFS: מערכת קבצים על מצבור של שרתים, אשר תוכננה מתוך ידע מדוקדק על צרכי השימוש בה, עם התאוששות מנפילות באופן אוטומטי וחלק.

GFS: שימוש ייעוצי

- קבצים גדולים מאוד $100 \text{ MB} \leq$
- קריאות גדולות ברצף (streaming) $1 \text{ MB} \leq$
 - Read once
- כתיבות סדרתיות גדולות שמבצעות append (כתיבה בסוף)
 - Write once
- על-ידי כמה לקוחות בו-זמנית!
 - אטומיות על תור producer-consumer ללא סנכרון

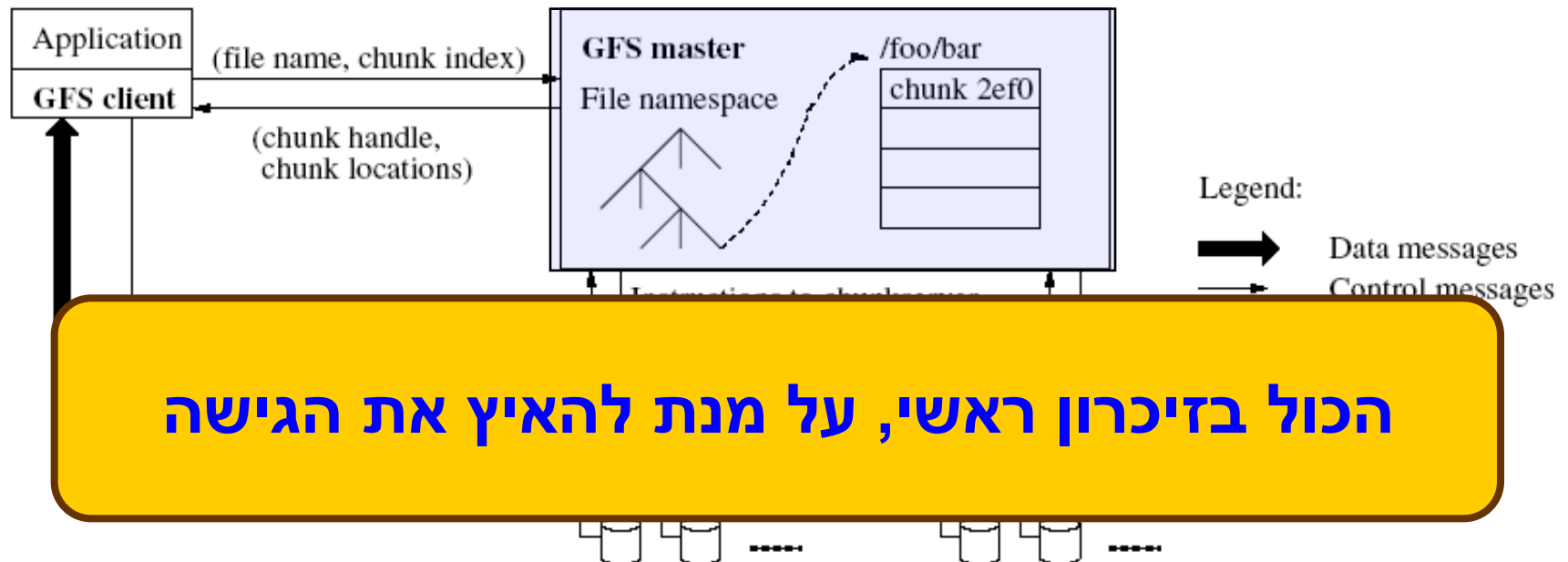
GFS: ארכיטקטורה

- שרת ראשי יחיד (עותקים משוכפלים לגיבוי)
- מאות / אלפי שרתי חתיכות (chunk servers)
 - חתיכה: חלק בגודל 64 MB של קובץ, עם מזהה יחודי
- המון לקוחות ניגשים לאותו קובץ או לקבצים שונים במצבור



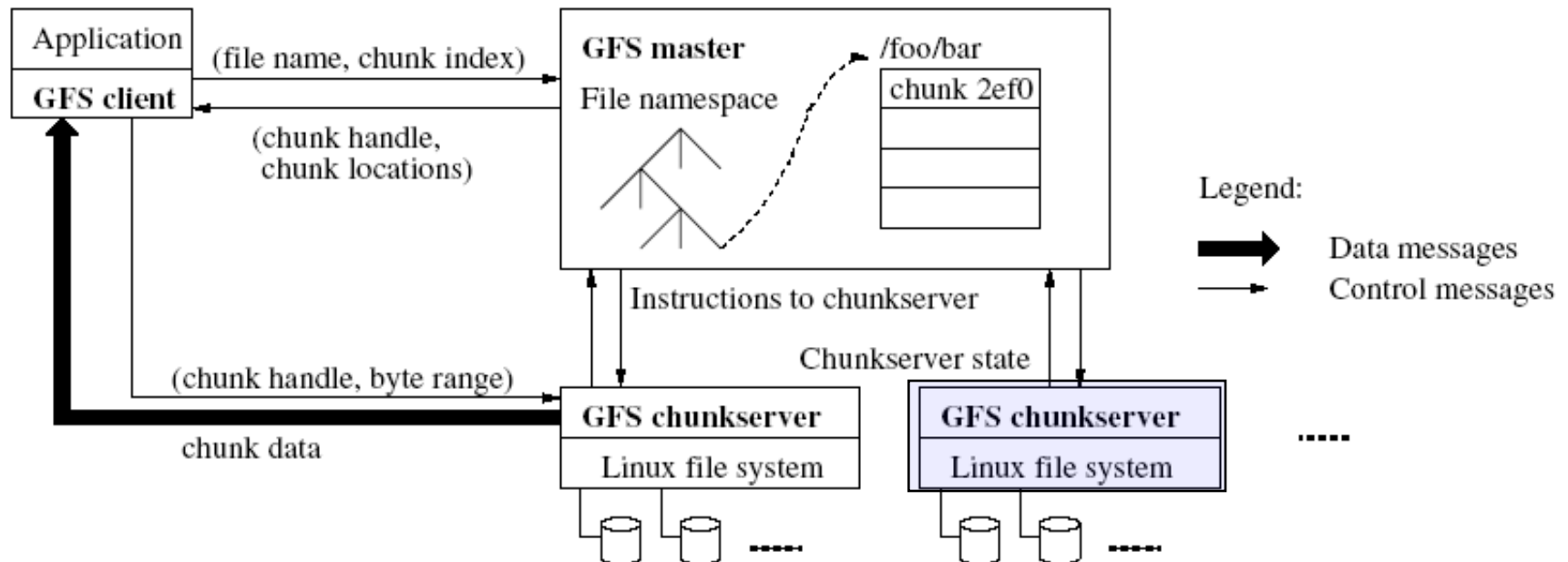
GFS: השרת הראשי

- מחזיק את כל ה metadata: מרחב השמות (מדריכים), הרשאות גישה, מיפוי מקבצים לחתיכות, מיקום החתיכות (על שרתי חתיכות)
- מנהל מתן הרשאות (leases) על חתיכות לשרתי החתיכות
- מעביר חתיכות בין השרתים



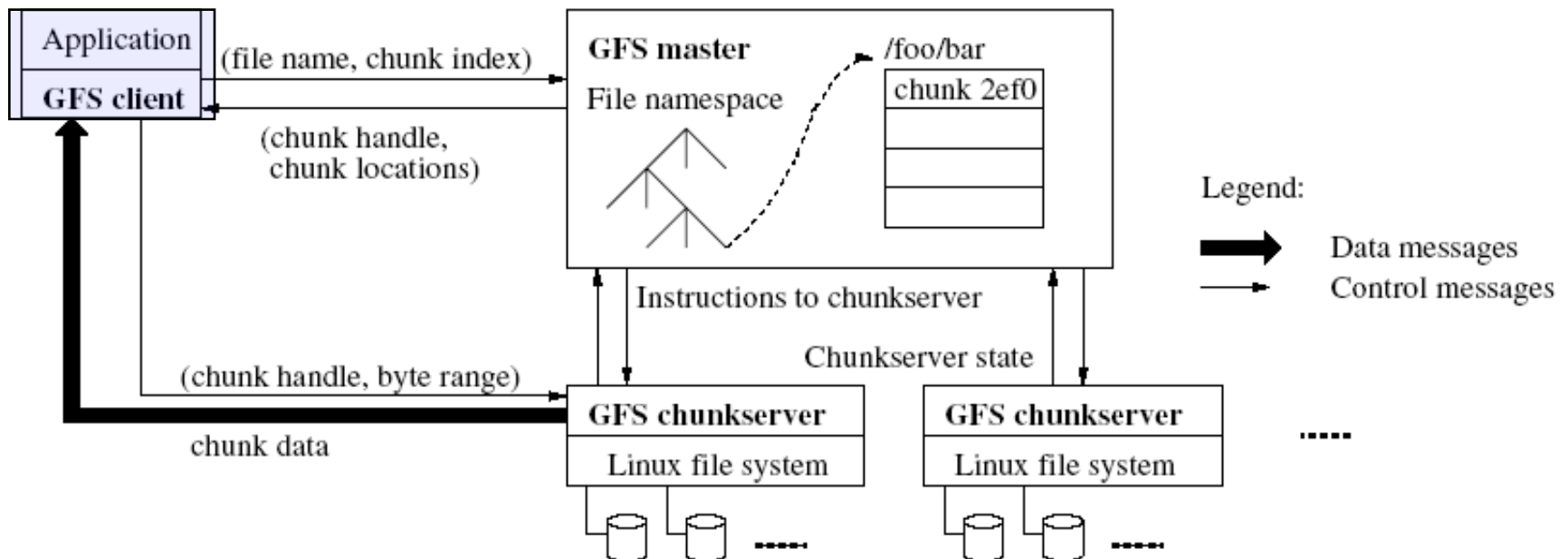
GFS: שרת חתיכות

- מאחסן חתיכות על דיסק מקומי, בשימוש ב Linux רגיל
- בקשות קריאה / כתיבה מציינות מזהה חתיכה וטווח בתים.
- חתיכות משוכפלות בכמה שרתי חתיכות (בדרך-כלל, 3)
- אין caching מיוחד



GFS: אקונו

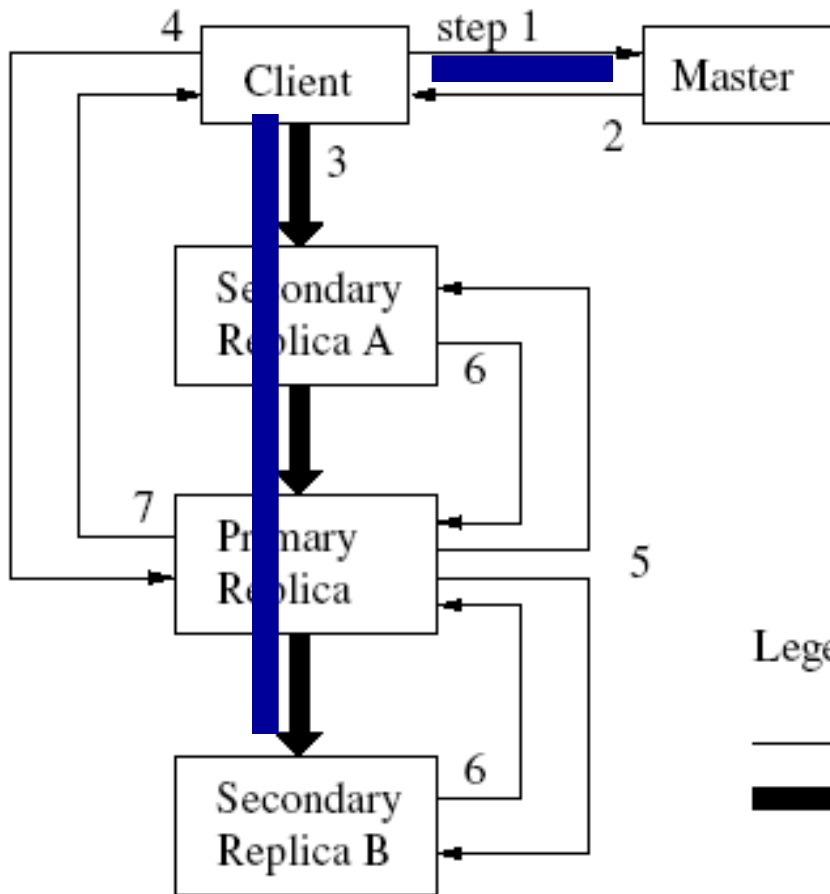
- שולח בקשות metadata אל השרת הראשי, ושולח בקשות לחתיכות אל שרתי החתיכות (קצת דומה לשרתי שיתוף קבצים)
- שומר metadata במטמון, לא שומר data במטמון
- חוסך בעיות של קונסיסטנטיות
- מטמון לא מועיל הרבה כאשר קוראים / כותבים פעם-אחת



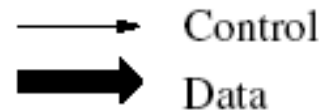
קריאה של הלקוח

- הלקוח מבקש מהשרת הראשי לקרוא (שם קובץ, אינדקס חתיכה)
- השרת הראשי משיב עם מזהה חתיכה, מיקומי החתיכה
- הלקוח מבקש את החתיכה משרת החתיכות "הקרוב ביותר" אשר מחזיק אותה
- נקבע על-פי כתובת ה IP
- שרת החתיכות שולח את החתיכה

כתיבה של לקוח 1



Legend:



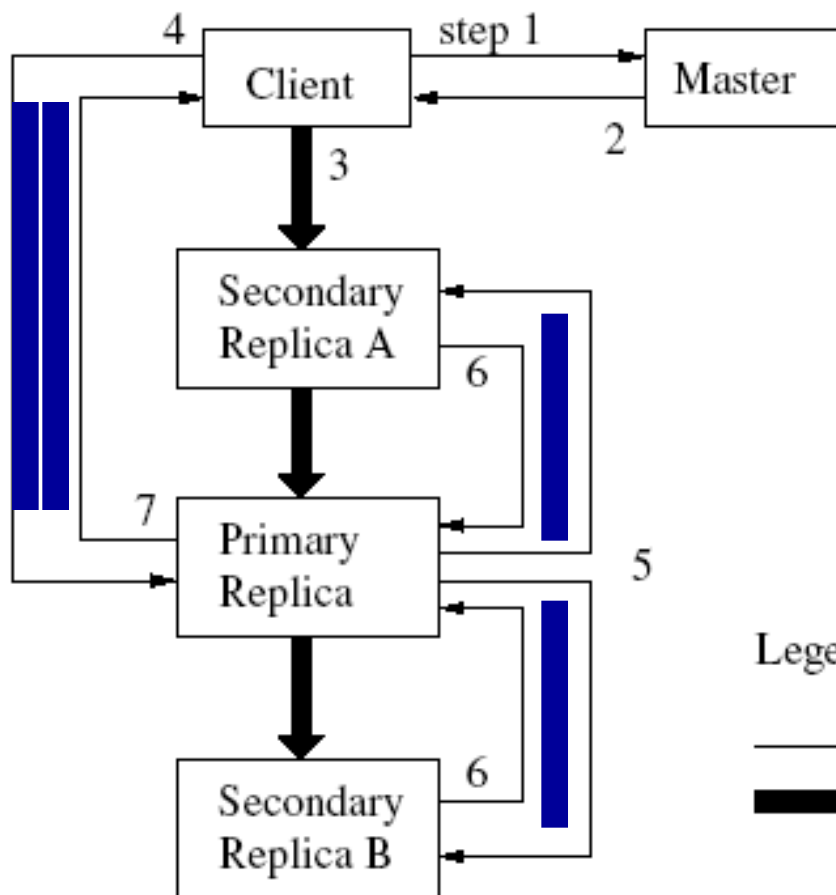
□ לכל חתיכה יש שרת חתיכות אחראי

■ מקבל lease מהשרת הראשי, שצריך לחדש אחרי מעט זמן (דקה)

□ הקליינט לומד מהשרת הראשי מי העותק האחראי והעותקים המשניים של כל חתיכה

□ שולח אליהם בקשות (הם מעבירים אחד לשני בשרשרת)

כתיבה ללקוח 2



Legend:




- כל העותקים מאשרים את הכתיבה ללקוח
- הלקוח שולח בקשת כתיבה לעותק האחראי
- השרת האחראי ממספר את בקשת הכתיבה, ומסדר אותה
- השרת האחראי מעביר את הבקשה לשרתים המשניים
- השרתים המשניים מאשרים את ביצוע הכתיבה
- השרת הראשי עונה ללקוח

עזר אספקטים

- מימוש יעיל במיוחד לשרשור (קובץ גדול במיוחד משמש כחוצץ בין יצרנים ולקוחות)
- התנהגות לא-פשוטה כאשר יש פעילות בו-זמנית על קבצים
 - שינויים ב metadata הם אטומיים (מבוצעים בשרת הראשי)
 - לשינויים ב data מובטחת אטומיות רק במקרים מיוחדים
- logging לגיבוי השרת הראשי

GFS: סיכום


הצלחה: גוגל משתמשת בו לחיפוש ולאפליקציות נוספות 


- זמינות ושרידות על חומרה זולה

- תפוקה טובה, בזכות ההפרדה בין בקרה (metadata) ובין מידע (data)

- תמיכה בכמויות מידע עצומות ו"הדבקה" בו-זמנית

- רעיונות דומים במערכת הקבצים Hadoop (שהיא קוד פתוח)

התנהגות לא-שקופה לאפליקציות (סמנטיקה לא-נקייה) 

ביצועים לא טובים לחלק מהאפליקציות 

- למשל, אם חוזרים וקוראים או כותבים מאותן כתובות, בגלל שאין מטמון אצל הלקוח

קלט פלט

באופן כללי, ניתן לחלק את התקני הקלט\פלט לשתי קטגוריות:

□ התקן בלוקים Block Devices התקנים המאחסנים מידע בבלוקים בגודל קבוע. לכל בלוק ניתן להתייחס לפי כתובת הבלוק, ולכן ההתייחסות לבלוק מסוים יכולה להתבצע באופן בלתי תלוי בהתייחסות לשאר הבלוקים. לדוגמה, דיסק קשיח ו- CD-ROM הם התקני בלוקים.

□ התקנים תווים Character Device פועלים עם רצפים של תווים. לא ניתן לדרוש מהתקן תווי לקבל תו מסוים מכיוון שאין לתווים כתובות, ולכן לא ניתן לבצע פעולת חיפוש כמו שניתן לבצע על בלוק בהתקן בלוקים. מדפסות, עכבר, כרטיסי תקשורת הם דוגמאות להתקנים תווים.

מערכת ההפעלה מפעילה כל התקן באמצעות כרטיס אלקטרוני מיוחד הנקרא בקר **Controller** הבקר מציג בפני מערכת ההפעלה ממשק לניהול כל פעולות הקלט/פלט האפשריות בהתקן זה. אוסף הפקודות של הבקר יוצר ממשק שבאמצעותו יש לפנות אליו, וברוב המקרים ממשקים אלה כפופים לסטנדרטים. פניה לבקר נעשית בעזרת תוכנת דרייבר דרך יציאות קלט פלט שהבקר מחובר אליהן (PORT).

שיטות לשיפור הביצועים:

- Disk Cache – אזור בזיכרון ששם נמצאים הבלוקים האחרונים שהשתמשנו בהם (לפי תדירות), מתוך נחה שאני אשוב ואשתמש בהם. בצורה כזאת אין צורך לשלוף אותם שוב מהדיסק. זהו רכיב חשמלי, ולכן מהירות הגישה גבוהה בהרבה.
- Free behind & Read ahead – נניח כי יש לי צורך בבלוק 17. אם כך רוב הסיכויים שנזדקק גם לבלוק 18 ו-19, ולכן רצוי לקרוא בלוקים עוקבים ולא רק בלוק אחד. כלומר הכנה מראש. בדומה רצוי לשחרר מראש בלוקים.
- RAM disk – (virtual disk) דיסק שנמצא ב-RAM (זיכרון ראשי). כותבים וקוראים לשטח כאילו זה הדיסק. חיסרון: כאשר החשמל נופל הכל הולך לאיבוד. יתרון – מהירות. היום שימוש זה כבר לא נפוץ. לא אמין.
- Disk Track buffer – מטמון בבקר הדיסק המשמש לקראת track שלם בבת אחת (החל מהסקטור בו נמצאים). חסכון בזמן רוטציה ובזמן העברה.
- Cluster – שמירת מקטעים בקבוצות (ע"י מ.ה.) כדי להמעיט בהזזות ראשי קריאה/כתיבה.

1. Port – device מתקשר עם המחשב ע"י שליחת אותות דרך כבל או אפילו דרך האוויר. התקשורת של ה-device עם המחשב דרך נקודת תקשורת נקרא .
2. Bus – כאשר התקן אחד או יותר משתמשים בקו נתונים משותף, התקשורת נקראת bus. במושגים יותר מדויקים, Bus הוא אוסף של קווים המגדירים פרוטוקול המציין קבוצות הודעות שניתן לשלוח בקווים.
3. Controller – אוסף של אלקטרוניקה המפעיל port, bus או device. לכל device ניתן לפנות בשני דרכים:
 1. פקודות I/O ישירות.
 2. שימוש ב-memory mapped I/O – הרגיסטרים של הבקר ממופים לתוך מרחב הכתובות של המעבד.

שיטות קלט-פלט

- תשאול Polling** – בכל שלב בודקים מי צריך את ה-CPU.
- ה-host מחכה עד שה-busy bit יהיה ב-0, ואז שולח פקוד (דרך command register), ומשנה את ה-ready bit ל-1. הבקר מעדכן את ה-busy bit ובודק את הפקודה.
- השלב שבו ה-host ממתין נקרא Polling. שיטה לא טובה משום שהדיסק כל הזמן עובד – כל הזמן מתבצעת בדיקה.

פסיקות Interrupt Driven IO, PIO: הבקר מודיע למעבד בעזרת קו תקשורת מיוחד בפס שקרא אירוע שמצריך תקשורת עמו. המעבד מגיב בהפעלת שגרת פסיקה של מערכת ההפעלה שמטפלת באירוע.

כשהתקן מהיר, פסיקות תכופות מידאי

גישה ישירה לזיכרון Direct Memory Access DMA: מערכת ההפעלה מורה לבקר להעביר בלוק גדול של נתונים בין התקן חיצוני (דיסק, רשת) ובין הזיכרון. הבקר מעביר את הנתונים ללא התערבות נוספת של המעבד.
חסרונות: חומרה מורכבת, תפיסת BUS שמונעת מ CPU להגיע ל RAM, צורך בנעילת דפים בזיכרון.

שיטות שונות לקריאות מידע מדיסק-תיזמון

• FCFS - First come First Served - קבלת מידע סדרתית. הולכים לפי הסדר. עפ"י הדוגמא

• SSTF - Shortest Seek Time First – מחפשים את הבקשות אם seek time מינימלי.

ממיינים את הרשימה והולכים לפי הרשימה הממוינת, כאשר המיון יעשה לפי seek time מינימלי.
חסרונות:

1. על כל בקשה שמגיעה צריך להכניס אותה לרשימה בצורה ממוינת. **אבל**, מיון הרשימה מהיר יחסית לתנועת הראש.

2. **הרעבה – Starvation** – מצב שבו Process לא מקבל תשובה לעולם. לא ניתן בוודאות

לקבוע מצב כזה, כי ייתכן ומערכת ההפעלה מייד מתפנה ל-process.

במקרה הנ"ל, הרעבה תיתכן כאשר יש process שדורש מידע מצילינדר מאוד מרוחק, וכל הזמן נכנסות לי בקשות לצילינדרים קרובים. במקרה זה עשויה הרעבה.

• SCAN – זרוע הדיסק נעה מהנקודה בה היא נמצאת לכיוון ההתחלה ומשם אל הסוף. בכל פעם שעוברים דרך צילינדר בודקים האם צריך לקרוא אותו ואם כן קוראים. פתרנו את בעיית ההרעבה. חסרונות:

1. נניח שאני בצילינדר 1 ומתחילה לנוע הלאה ובדיוק מגיעה בקשה לצילינדר 1. אומנם לא תהיה הרעבה אבל ייקח זמן עד שאני אגיע אליו חזרה. השיטה דוגלת באחידות אבל לא ביעילות.
2. הזרוע כל הזמן נעה מקצה אחד לשני, ובכל תזוזה קוראים. נניח שאני נמצאת בצילינדר 2 והתקבלה כרגע בקשה למידע מצילינדר 1 ו-4. במצב כזה המידע המאוחר יותר (שנמצא בצילינדר 4) יקרא לפני המידע "המוקדם". הבעיה היא שלא נקרא את המידע לפי הסדר.
3. הראש נע מהצילינדר ה-0 ועד הצילינדר 199, כאשר ייתכן והמידע הדרוש מצוי בין צילינדרים 50-100.

• C-SCAN – שיפור השיטה SCAN: הראש נע מהמיקום הנוכחי אל הצילינדר האחרון, וכאשר מגיעים אליו, מוזזים את הראש ישירות לצילינדר ה-0, ומשם שוב הולכים לסוף. כלומר מבצעים קריאה מצילינדר רק כאשר מתקדמים קדימה.

• LOOK – שיטה זו דומה ל-SCAN, אלא שהתנועה היא בין הצילינדר הימני ביותר המבוקש (ולא המינימלי בדיסק) לבין השמאלי ביותר. פתרון חסרון 3 של השיטה SCAN.

• C-LOOK – בדומה ל-C-SCAN: קריאה רק כאשר מתקדמים קדימה.

Disk Reliability

הדיסק הוא יחידה עם מנגנונים מכניים, ועם חלקים נעים, ייתכן נפילות. נפילת דיסק גורמת לאובדן מידע רב. השחזור לוקח זמן רב ולא תמיד אפשרי בשלמות.

נעשו מספר שיפורים בטכניקות השימוש בדיסקים. שיטות אלו כוללות שימוש במספר דיסקים העובדים במקביל. לצורך הגברת המהירות, disk stripping מתייחס לקבוצה של דיסקים כאל יחידה אחת. כל בלוק נתונים מפורק לתת בלוקים שכל אחד מהם נשמר בדיסק אחר. הזמן הנדרש להעברת בלוק לזיכרון השתפר בצורה משמעותית, משום שכל הדיסקים מעבירים את הבלוקים שלהם בצורה מקבילית.

יתרון:

הרבה דיסקים קטנים וזולים במקום דיסק אחד גדול ויקר.

חסרון:

העברת הרבה יחידות קטנות במקביל – גישה איטית יותר. פתרון עשוי להיות ע"י העברת יחידה גדולה במקביל לזיכרון אם יש דיסקים מסונכרנים).

RAID

ארגון מסוג זה נקרא בדר"כ Redundant array of inexpensive disks. בנוסף, שיטות RAID רבות שיפרו את האמינות ע"י כפילות (redundancy) נתונים.

שיכפול המידע במערך של דיסקים (קטנים וזולים) על מנת להבטיח אמינות.

• RAID level 0 – מערך של דיסקים (disk stripping).

• RAID level 1 – נקרא גם mirroring או shadowing. תמיד יהיו 2 עותקים מכל דבר. כל מידע נכתב גם לעותק. במקרה של נפילה תמיד ניתן לשחזר מהעותק.

אמינות גבוהה. חסרונות: מספר כפול של דיסקים, פעולת כתיבה לוקחת זמן כפול.

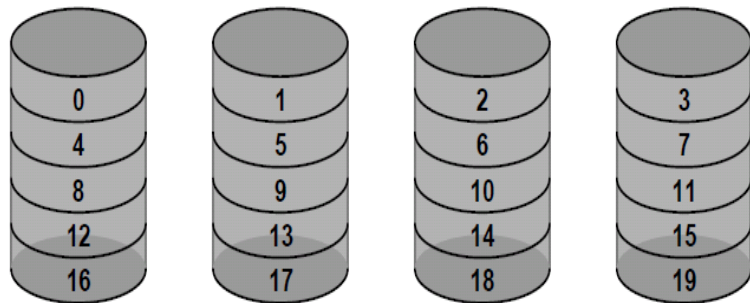
• RAID level 2 – memory style error correcting code. שימוש בקוד המתקן שגיאות ברמת הבתים.

יתרונות: מאוד אמין, פחות דיסקים מהגישה הקודמת.

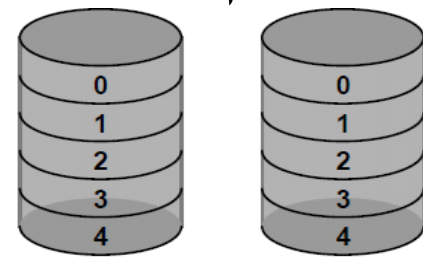
• RAID level 3 – Byte-interleaved parity. קיים דיסק parity יחיד המתקן שגיאה אחת. כל בית בדיסק מיועד לכל יתר הדיסקים (כל בית לדיסק). בזמן הכתיבה מחשבים את הזוגיות של הבתים הנמצאים בכל הדיסקים באותה הכתובת, וכותבים את bit הזוגיות באותה כתובת בדיסק הנוסף. ברגע שדיסק אחד הלך או בלוק אחד הלך ניתן לשחזור אותו עפ"י ה-parity bit. חסרונות: כתיבה דורשת גישה לכל הדיסקים.

• RAID level 4 – Block-interleaved parity. הפעם מקצים בלוק לכל דיסק ולא בית. כתיבה מחשבים בלוק parity ולא ביט בודד.

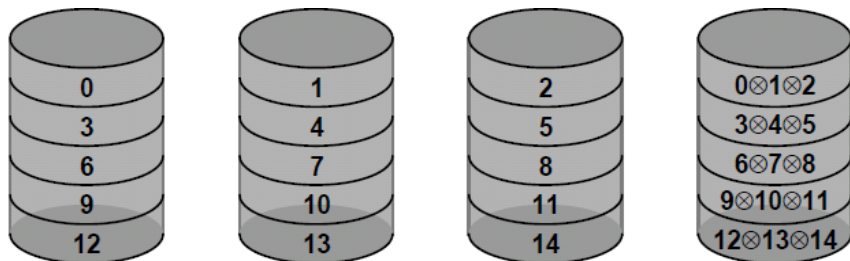
• RAID level 5 – Block-interleaved distributed parity. בלוק ה-parity נשמר בכל הדיסקים ולא בדיסק בודד.



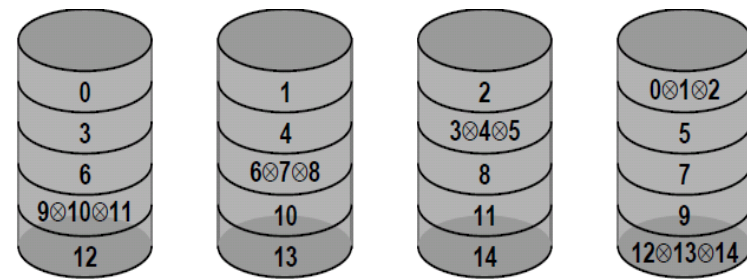
RAID 0



RAID 1



RAID 3



RAID 5

וירטואליזציה

מכא

□ מערכת ההפעלה מספקת אשליה של "מכונה וירטואלית" לתהליכים

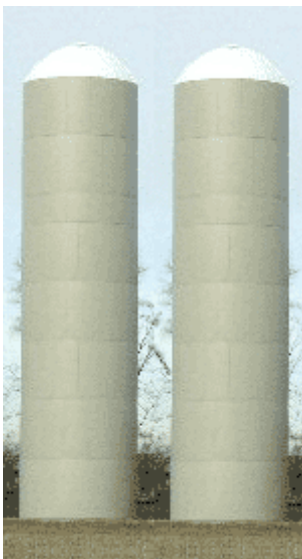
- זיכרון וירטואלי השייך כולו לתהליך
- "מעבד וירטואלי" שבו רצים רק חוטי התהליך
- ערוצי תקשורת ל"מכונות וירטואליות" אחרות

□ אבל..

■ אין בידוד מלא בין תהליכים:

- ניתן ללמוד על תהליכים וחוטים אחרים דרך קריאות מערכת
- המשאבים משותפים - בד"כ אין הקצאה קשיחה של משאבים לתהליכים
- פירצת אבטחה במערכת ההפעלה יכולה להזיק להרבה תהליכים
- קשה לפתח תוכנה "רגישה" כגון מנהלי התקנים ועדכונים לגרעין מ"ה
- תלות בחומרה מדור מסוים, בקונפיגורציה מסוימת

מכונה ייצודית לכל יישום?



בידוד מרבי בין יישומים 👍

אפשר להקדיש מכונה לפיתוחים בגרעין 👍

עדיין לא מבטל את התלות בחומרה.. 🙄

מביא לממגורות מידע (data silos)

□ בזבוז רב של יכולות חומרה (חישוב, אחסון, ק/פ)

■ קשה לשתף ביעילות משאבים בין מכונות נפרדות

□ עלות גבוהה של ניהול: התקנת כל מכונה, קישוריות, תחזוקה...

וירטואליזציה

□ מפרידים בין הזוג [מערכת הפעלה, אפליקציה/שירות] לבין החומרה.

□ ממתגים שירותים דלילים על חומרה משותפת.

□ מעבירים ממכונה למכונה לפי הצורך.

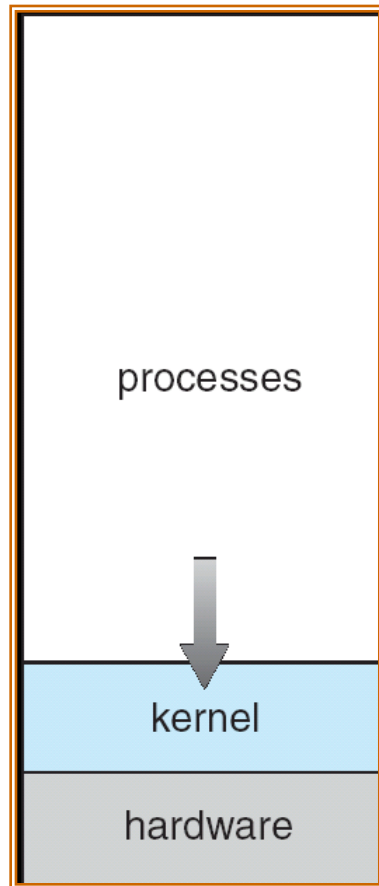
□ מתקינים זוגות [מערכת הפעלה, אפליקציה/שירות] לפי הצורך.

■ שירותים חדשים

■ בדיקת שדרוגים ושינויים

■ שימושים ניסיוניים

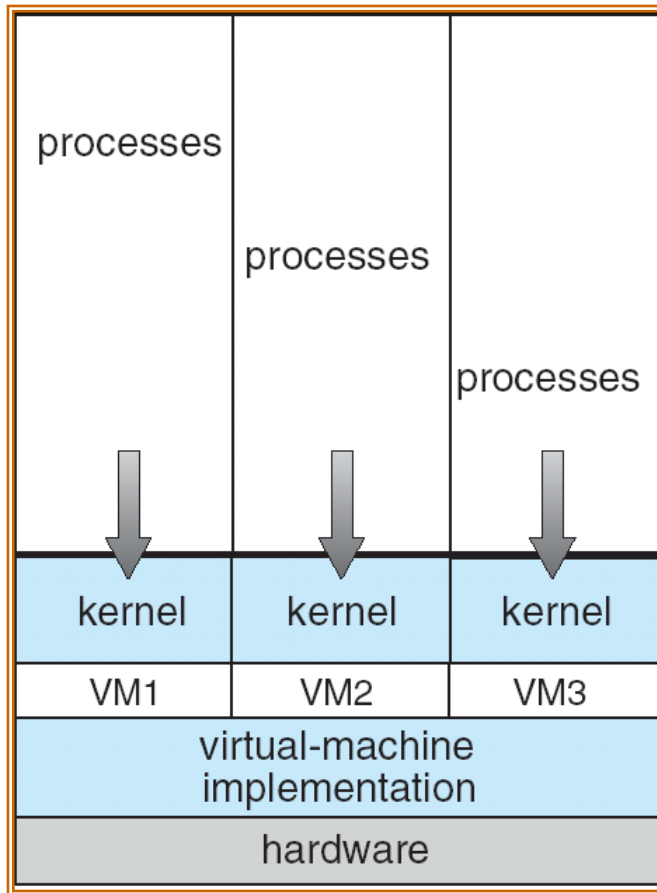
Virtual Machine Monitor (VMM)



Non-virtual Machine

- שכבת תוכנה (שכבת הוירטואליזציה)
בין מערכת ההפעלה לחומרה
 - נקראת גם hypervisor
 - מדמה חומרה "אמיתית" כלפי מערכת ההפעלה (התקנים, רגיסטרים, קלט / פלט...)
- ניתוק התלות בין מערכת ההפעלה לחומרה
 - מערכת ההפעלה "חושבת" שהיא רצה על חומרה אמיתית
 - למעשה רצה על מכונה וירטואלית (virtual machine)

Virtual Machine Monitor (VMM)



Virtual Machine

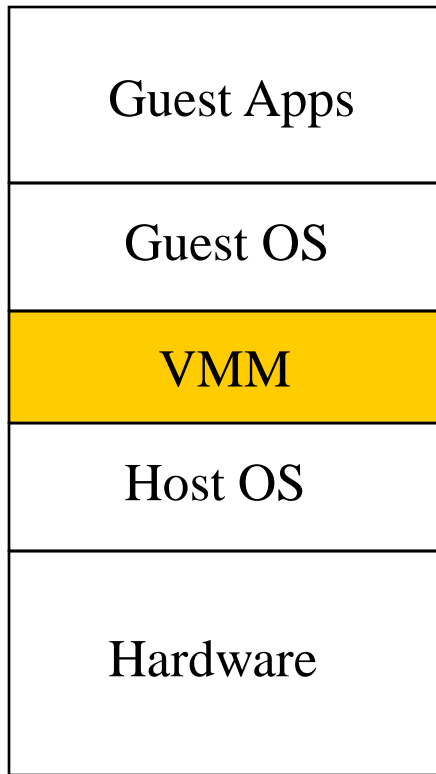
□ שכבת תוכנה (שכבת הוירטואליזציה)
בין מערכת ההפעלה לחומרה

- נקראת גם hypervisor
- מדמה חומרה "אמיתית" כלפי מערכת ההפעלה (התקנים, רגיסטרים, קלט / פלט...)

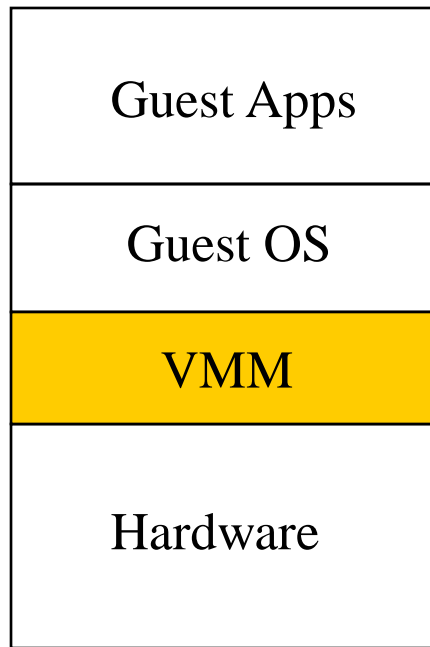
□ ניתוק התלות בין מערכת ההפעלה לחומרה

- מערכת ההפעלה "חושבת" שהיא רצה על חומרה אמיתית
- למעשה רצה על מכונה וירטואלית (virtual machine)

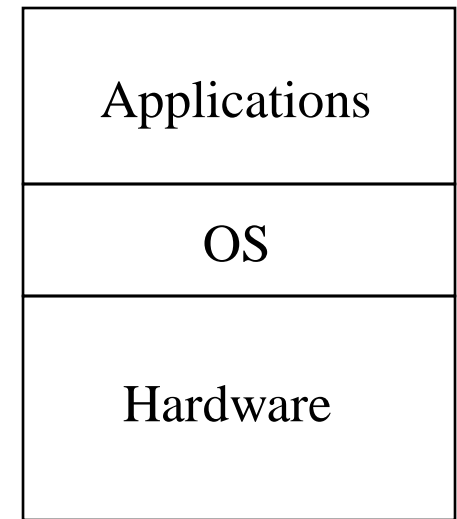
סוגי מכונות וירטואליות



User-mode virtualization

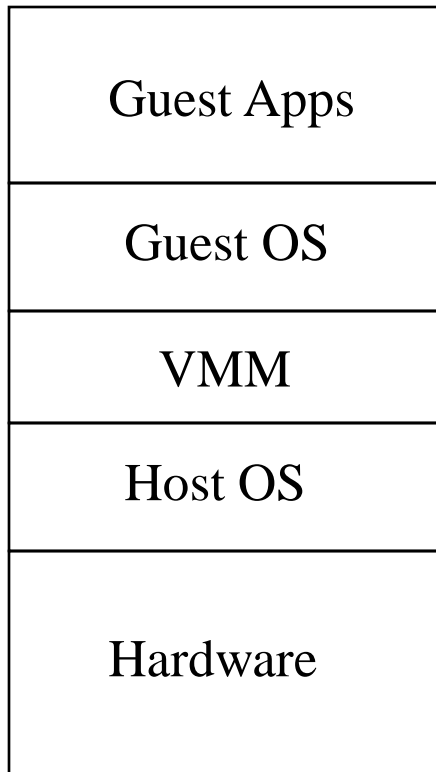


Native virtualization



מערכת הפעלה מסורתית

טראַנינאָואַציע



□ מערכת הפעלה מארחת רצה על החומרה

■ יחד עם ה VMM מדמה את הסביבה עבור...

□ מערכת ההפעלה האורחת, אשר רצה על הסביבה המדומה

■ אותה אנחנו רוצים לבודד

User-mode virtualization

האתגר

- קשה להבטיח למערכת ההפעלה האורחת העתק זהה למכונה המדומה
- לדוגמה: טבלאות דפים ופסיקות דף
 - אם למע"ה האורחת יש טבלאות דפים משלה, אז ה VMM צריך
 - להעתיק טבלאות אלה
 - לעקוב אחרי פניות ועדכונים לטבלאות ולסמלץ אותם
 - כשיש פסיקת דף (page fault)
 - ה VMM צריך להחליט אם היא שייכת למע"ה האורחת או לו
 - אם שייך למע"ה האורחת צריך לסמלץ את פסיקת הדף עבודה
- באופן דומה, ה VMM צריך לתפוס ולסמלץ כל פעולת מכונה מוגנת (privileged)

לא תמיד אפשרי

יש אספקטים של החומרה שאינם ניתנים לוירטואליזציה

- רגיסטרים מסוימים שאינם חשופים

- התקנים מיוחדים

- שעונים, והתנהגות זמן-אמת

נפתרים על-ידי כלים ו drivers במע"ה האורחת

- *VMware Tools*

סוגים של וירטואליזציה

□ וירטואליזציה מלאה (Full Virtualization)

- שקיפות - יכולה להריץ מ"ה ללא שינוי
- ניתן לשפר ביצועים בסיוע החומרה (AMD-V, Intel VT)
- VMware, QEMU, Xen

□ פארא-וירטואליזציה (Para-Virtualization)

- מערכת ההפעלה האורחת מודעת ל-VMM (באמצעות מנהלי ההתקנים)
- מאפשר שיפור ניכר בביצועים על חשבון השקיפות
- Xen, Z/VM

□ מבוסס מערכת הפעלה (OS-Level Virtualization)

- כל תהליך או קבוצת תהליכים רצים בתוך "מיכל" (Container) שמתנהג כמו מחשב נפרד לכל דבר (קונפיגורציה נפרדת, רשת, מערכת קבצים)
- שיתוף הגרעין וניהול המשאבים בין המיכלים
- שיפור נוסף בביצועים, תלות נוספת בתאימות
- Solaris Containers, Virtuozzo Containers, POWER WPARs

יתרונות ומסכנות

✓ כמו במכונות ייעודיות..

- בידוד מלא בין יישומים
- הקצאה קשיחה של משאבים (נאכפת ע"י ה-VMM)
- אפשר לפתח ולדבג קוד גרעין בלי לחשוש מתקלות

✓ ... ואפילו יותר טוב

- ניצול גבוה של משאבים בהרצת כמה מכונות וירטואליות באותה מכונה
- חיסכון בעלויות התקנה וניהול
- נידוד מכונות וירטואליות בין מחשבים פיזיים לצורך שיפור ביצועים והתאוששות

✗ הוספת שכבה מתווכת בין התהליך לחומרה

✗ בעיות של בטיחות

✗ הגדלת התקורה של משאבים (זיכרון, זמן מעבד) שאינם בשימוש תהליכים (ומשמשים עבור ה-VMM + מספר מערכות הפעלה)

דואמאות בולטות

VMWare, VirtualBox □

■ מאפשרים לכמה מערכות הפעלה קלאסיות לרוץ מעל אותה חומרה

Xen □

■ קוד פתוח

■ מערכת ההפעלה האורחת מותאמת ל VMM

■ נתמך על מעבדי Intel, AMD

Java virtual machine □

■ החומרה שמדומה על-ידי ה VMM אינה חומרה קיימת

השוואה:

http://en.wikipedia.org/wiki/Comparison_of_virtual_machines_features

פדיונח



An iMac computer, with VMware Fusion, which enables it to run Windows XP Pro on the left screen, Windows Vista Home on the right, and Mac OS X Leopard in the background.

בהצלחה!