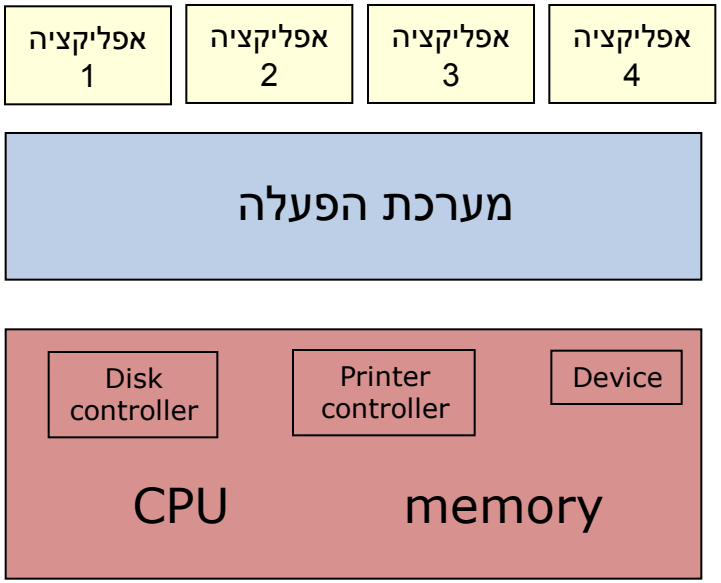
מהי מערכת הפעלה?

- שכבת תוכנה לניהול והסתרת פרטים של חומרת המחשב.
- מספקת לאפליקציה אבסטרקציה של מכונה וירטואלית ייעודית וחזקה (זיכרון עצום, מעבד ייעודי חזק מאוד...)
- מנהלת את משאבי המערכת ומשתפת אותם בין תהליכים, תוכניות, ומשתמשים.
- ממשק נוח למשתמש SHELL

תפקיד מערכת ההפעלה

- מאפשרת להריץ אפליקציות
- מבטיחה נכונות.
  - גבולות זיכרון
  - עדיפויות
  - מצב יציב
- מספקת נוחיות.
  - הסתרת פרטים
  - תיאום
  - קריאות מערכת-הפעלה
  - מערכת קבצים



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 המקבלים 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 כ**מכונה מדומה או כמכונה מורחבת.**

שיתוף 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.

התפתחות מערכות הפעלה

חומרה יקרה ואיטית, כוח-אדם זול
IBM S/360: 24x7, ניצול החומרה Batch jobs

חומרה יקרה ומהירה, כוח-אדם זול
Unix: Interactive time-sharing

חומרה זולה ואיטית, כוח-אדם יקר
MS-DOS מחשב אישי לכל משתמש:

- חומרה זולה מאוד, כוח חישוב רב.
  - ריבוי משימות,ריבוי משתמשים: 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

העתיד הקרוב

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/Ubiquituous Computing

מבנה המחשב

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

מנגנוני 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 מוגנות

- 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.
- 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.

יום בחיים 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
- Basicl/O System.בסיום האיתחולים,יטען טוען העליה את גוש העליה של מ.ה.(boot block),והוא יטען את יתר גרעין המערכת.בדרך כלל זה Master Boot Record-MBR- סקטור 0 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 מערכת ההפעלה

- תהליכים
- זיכרון
- קלט / פלט
- זיכרון משני
- מערכות קבצים
- הגנה
- ניהול חשבונות משתמשים
- ממשק משתמש (shell)

ארגון מערכת ההפעלה

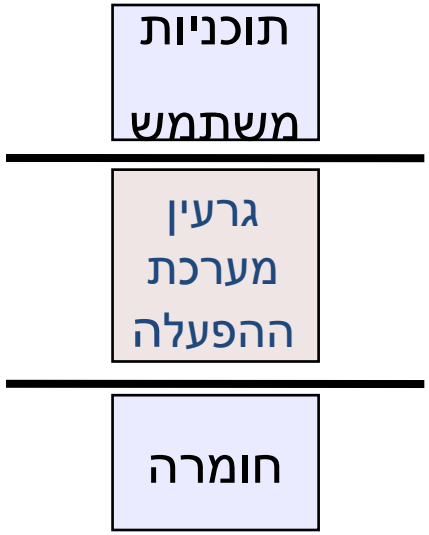
בראשית... מונוליתית

✓ תקשורת זולה בין מודולים

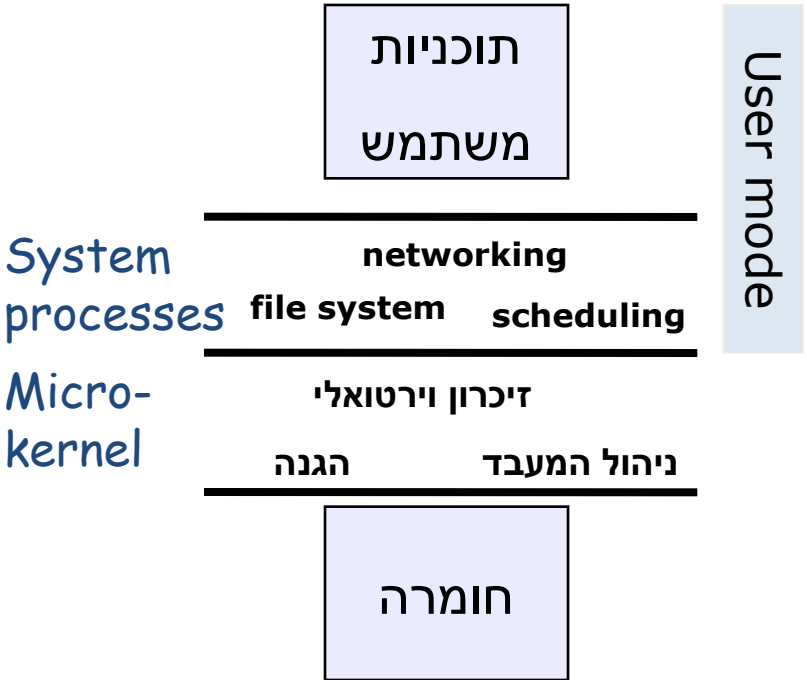
✗ קשה להבין

✗ קשה לשנות או להוסיף רכיבים

מה האלטרנטיבה?



אח"כ... גרעין קטן



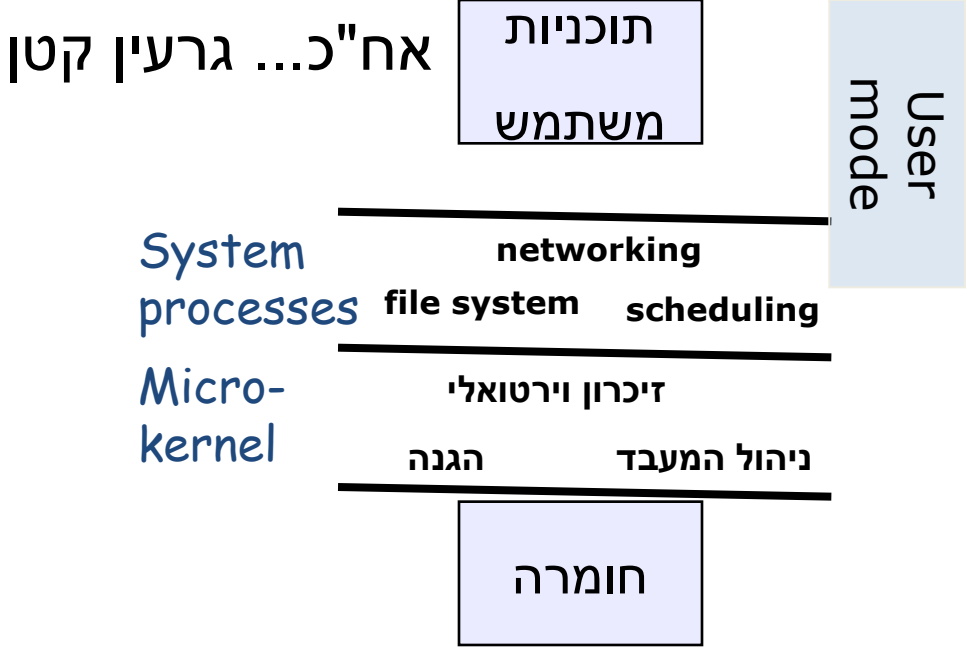
מערכת מונוליתית היא למעשה אוסף גדול של פונקציות שונות.

מערכות מרובדות ***-layered systems***

מערכות שמרכיביהן מאורגנים ברבדים אשר נמצאים ביחס היררכי. לכל רובד מוגדר ממשק שדרכו הרבדים הסמוכים בהיררכיה יכולים לתקשר ביניהם.

- Micro Kernel: שכבה דקה מספקת שירותי גרעין
  - ✓ אמינות גבוהה יותר
  - ✓ קל להרחיב ולשנות
  - ✗ ביצועים גרועים (מעבר בין user-mode לבין kernel-mode)

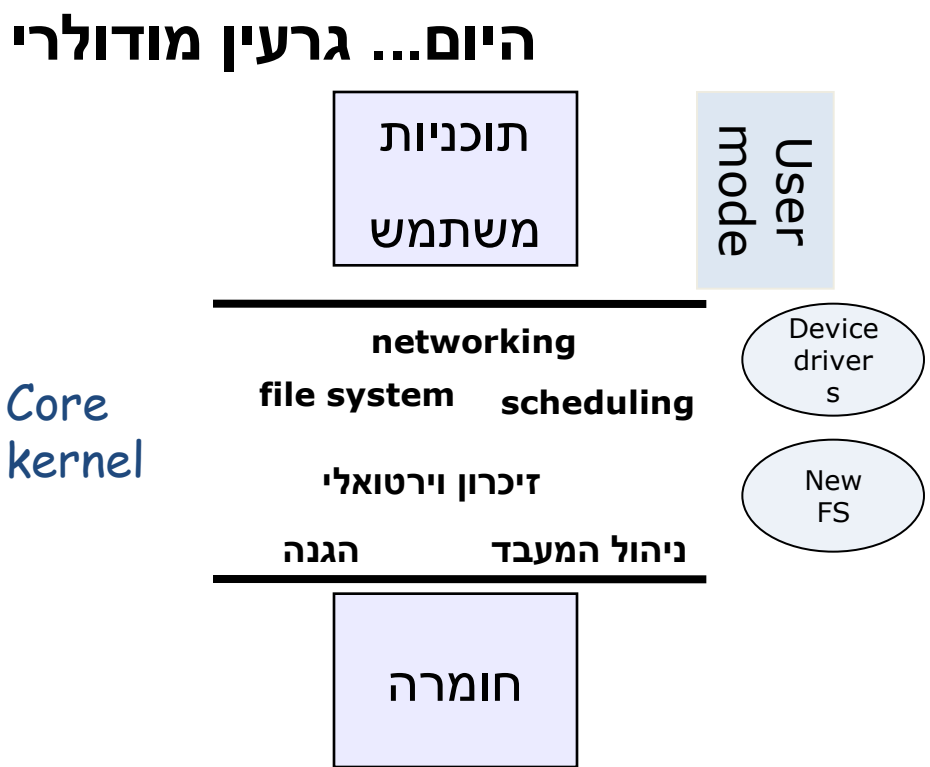
דוגמאות:
Mach, OS X, ~Windows NT



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)



virtual machine מכונה מדומה - אחד התפקידים של מערכתהפעלה 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.

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

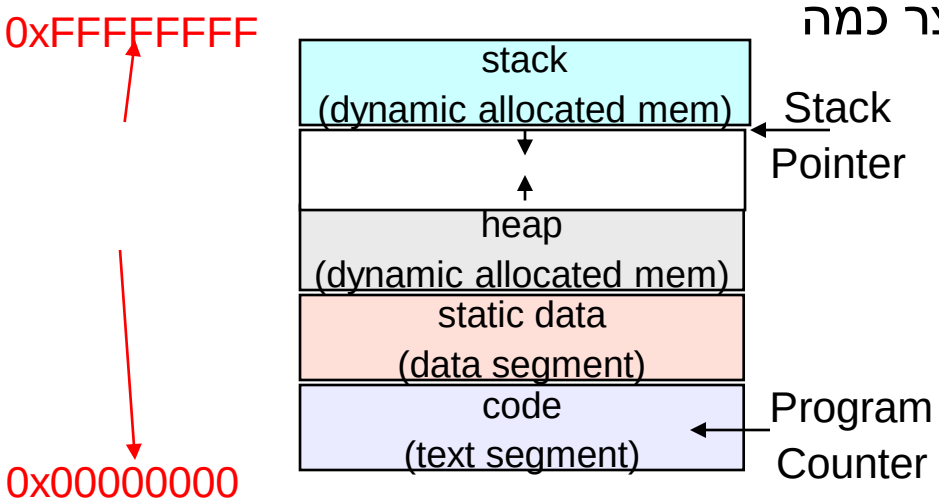
מה מאפיין תהליך?

- מרחב כתובות
- קוד התוכנית
- נתונים
- מחסנית זמן-ביצוע
- program counter
- רגיסטרים
- מספר תהליך (process id)

תהליך לעומת תוכנית:

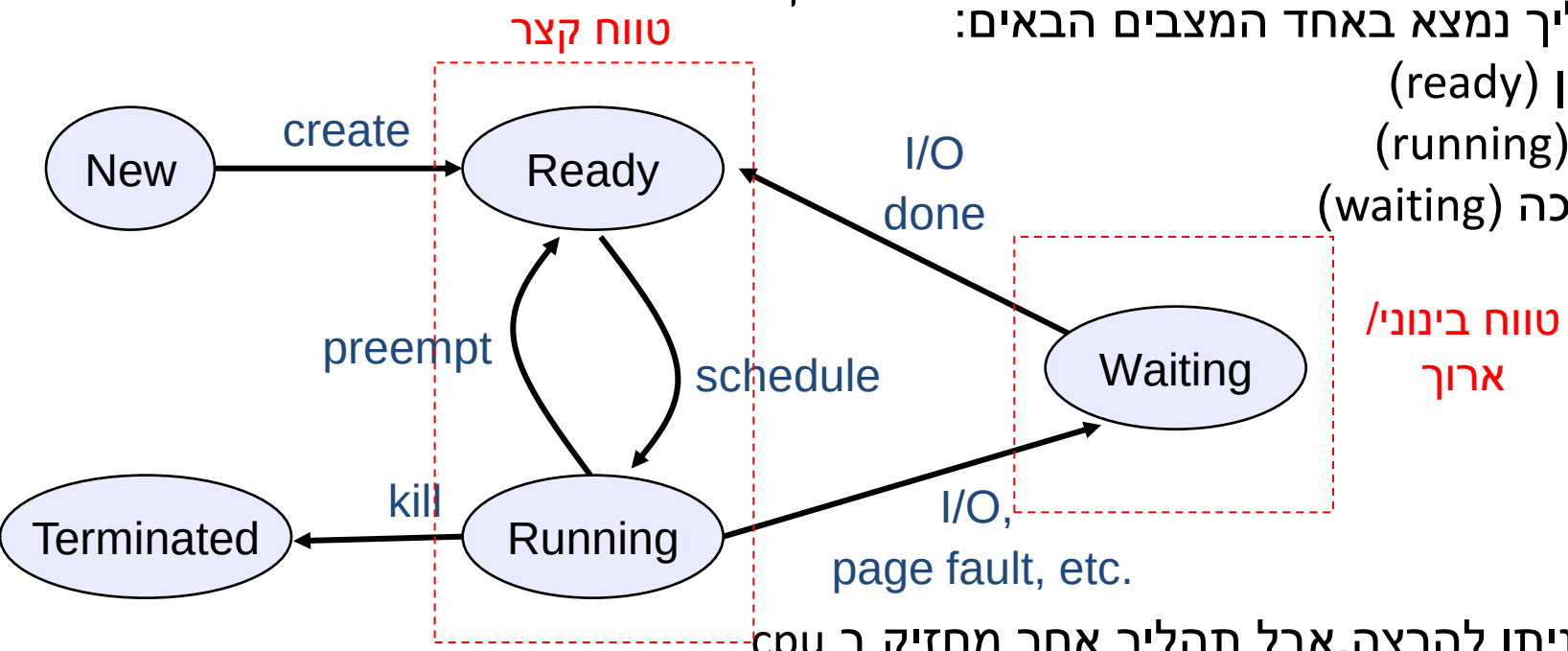
- תוכנית היא חלק ממצב התהליך
- תוכנית יכולה לייצר כמה תהליכים

מרחב הכתובות של התהליך



מצבי התהליך

- כל תהליך נמצא באחד המצבים הבאים:
  - מוכן (ready)
  - רץ (running)
  - מחכה (waiting)



- מוכן = ניתן להרצה,אבל תהליך אחר מחזיק ב cpu
- רץ = מתבצע בתוך ה cpu, כמה תהליכים יכולים להיות במצב זה?
- מחכה = ממתין שיקרה מאורע כלשהו, למשל input / output, ולא יכול להתקדם בינתיים

מבני הנתונים של תהליך

- 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.

Process id (PID)
Execution state
Program counter
Stack pointer
Registers
Memory limits
Scheduling priority
Username
Open files status

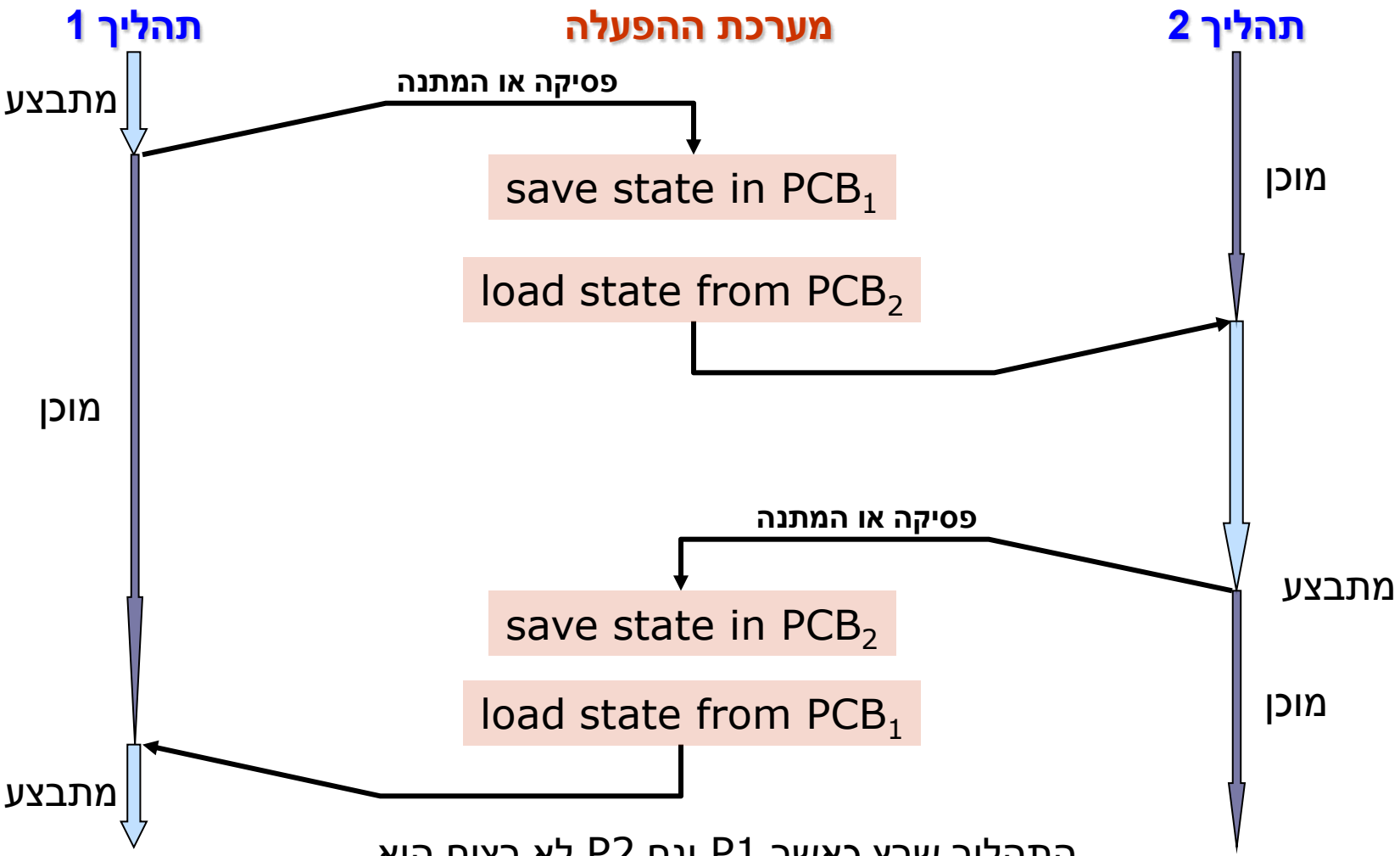
- כאשר תהליך רץ, המצב שלו נמצא במעבד:
  - PC, SP, רגיסטרים
  - רגיסטרים לקביעת גבולות זיכרון
- כאשר המעבד מפסיק להריץ תהליך (מעבירו למצב המתנה), ערכי הרגיסטרים נשמרים ב-PCB.
- כאשר המעבד מחזיר תהליך למצב ריצה, ערכי הרגיסטרים נטענים מה-PCB.

התהליך יכול לקרוא שדות מה PCB שלו רק באמצעות קריאות מערכת-הפעלה.

Context Switch: העברת המעבד מתהליך אחד לשני.

-

Context switch



התהליך שרץ כאשר P1 וגם P2 לא רצים הוא תהליך הזמן.
שימו לב לתקורה של מערכת ההפעלה.
איפה נמצא ה-PCB כאשר התהליך לא רץ?

ה-PCB ותורי המצבים

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()

- 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

```
int main(int argc,
         char **argv)
{
  int child_pid = fork();
  if (child_pid == 0) {
   printf("Son of %d is %d\n",
        getppid(),getpid());
    return 0;
  } else {
    printf("Father of %d is
                    %d\n",
       child_pid,getpid());
    return 0;
  }
}
```

מה יהיה סדר השורות בפלט? -- שים לב לערבובים!

לבן יש **עותקים** של המשתנים הגלובליים, ועכשיו יכול לשנות אותם!

נראה בהמשך מימוש יעיל יותר ל fork()

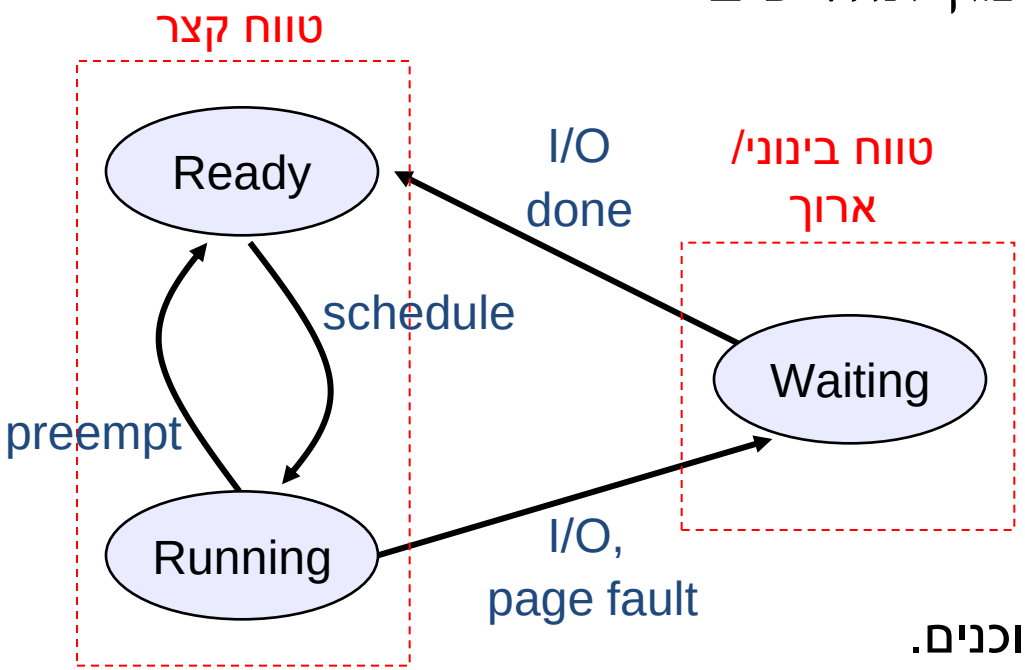
איך מפעילים תוכנית חדשה?

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

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

לא יוצר תהליך חדש!

זימון 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 בבודדת (למשל, אחרי המתנה למשאב שאינו ניתן לשיתוף כמו מדפסת)

מדדים להערכת אלגוריתם לזימון 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

אלגוריתם 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.

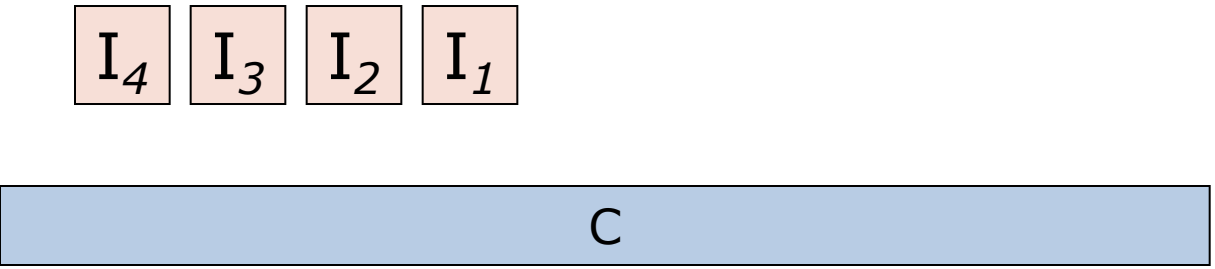
➢ מימוש פשוט: תור התהליכים המוכנים הוא FIFO.

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

זמן המתנה ממוצע = (0+24+27)/3 = 17.

זמן המתנה ממוצע = (0+3+6)/3 = 3.

אפקט השיירה Convoy



C תהליך עתיר חישובים

$I_1,\ldots,I_n$ 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 = 10-100msec.
- מימוש על-ידי timer שמייצר פסיקה כל q יחידות-זמן.

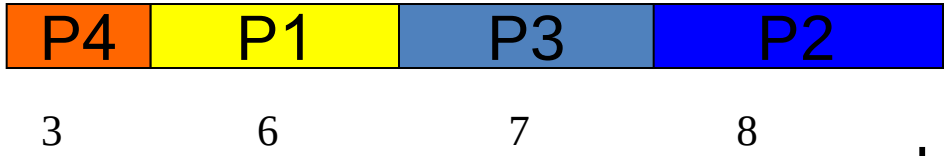
Shortest Job First (SJF)

נקרא גם Shortest Processing Time First

מריצים את התהליך עם זמן ביצוע מינימאלי, עד לסיומו.Non-preemptive .

– כל התהליכים מגיעים יחד. - זמן הביצוע של תהליך ידוע מראש.

P1 (6) P2 (8) P3 (7) P4 (3)



זמן המתנה ממוצע = (0+3+9+16)/4= 7

מינימאלי לכל סדר זימון אפשרי!

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.

מדדי יעילות למנגנוני זימון

T_i זמן השהייה 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

ניבוי זמן הריצה של תהליך

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

$$\tau_{i+1} = \alpha \cdot \tau_i + (1-\alpha) \cdot t_i$$
$$0 \le \alpha \le 1$$

- τ_i – הערכת זמן הביצוע לסיבוב ה-i
- t_i – זמן הביצוע בפועל בסיבוב ה-i
- $\alpha = 0$ ⇦ רק זמן הריצה האחרון קובע
- $\alpha = 1$ ⇦ זמן ריצה בפועל לא משפיע (ואז τ_0 קובע מאוד)

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).

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.

תהליכים-דייאט: חוטים עלות 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

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

במערכות הפעלה קלאסיות "חוט" יחיד בכל תהליך

במערכות הפעלה מודרניות תהליך הוא רק מיכל לחוטים

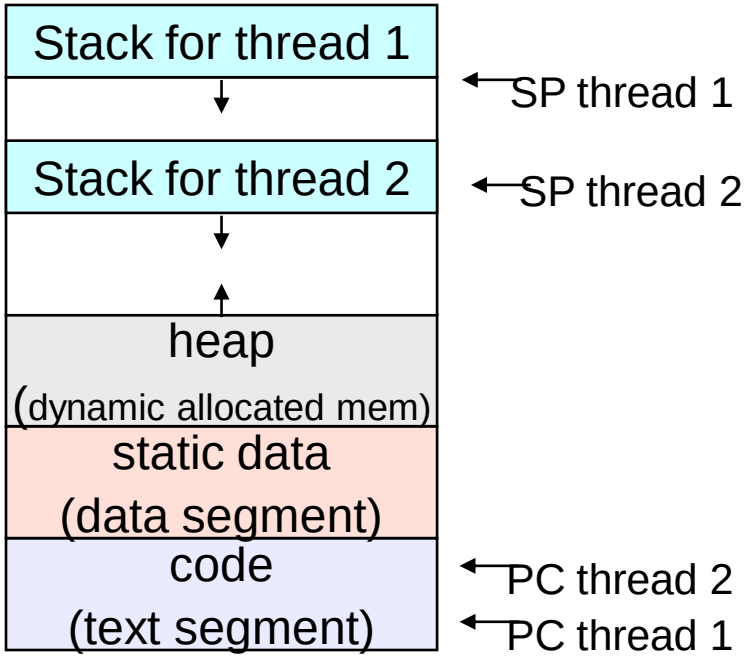
תהליכים-דייאט lightweight processes

משתפים מרחב כתובות, הרשאות ומשאבים

- לכל חוט מוקצים המשאבים הבאים:
  - program counter
  - מחסנית
  - רגיסטרים

נמצאים ב Thread Control Block (TCB)

מרחב הכתובות של תהליך מרובה-חוטים



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 מערכת (kernel threads)

מוכרים למערכת ההפעלה

נקראים lightweight processes

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

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?

תיאום

- תהליכים משתפים פעולה:מניעה הדדית או/ו תלות הדדי
  - גישה למשאבים משותפים, למשל זיכרון משותף (בעיקר חוטים).
  - העברת נתונים מתהליך אחד לשני דרך התקן משותף.
  - התקדמות של תהליך אחד תלויה בנתונים של השני.
- חייבים לתאם את השיתוף:
  - מניחים שביצוע התהליכים משולב באופן שרירותי.
    למתכנת האפליקציה אין שליטה על זימון התהליכים.
  - שימוש במנגנוני תיאום (synchronization).
- מימשנו פונקציה למשיכת כסף מחשבון בנק. race condition

```
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.

פיתרון 3 ☺

לא סימטרי

אם שניהם משאירים פתק בו זמנית

(race condition),

A יקנה חלב!

--לא הוגן -- רק לשני תהליכים ☹

```
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
```

קוד כניסה

קטע קריטי

קוד יציאה

שעה	לאה:	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

Peterson's Solution for 2 i and j

flag[i] = TRUE; turn = j;

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

CRITICAL SECTION

flag[i] = FALSE;

REMAINDER SECTION

flag[]- מוכנות(רצון) להיכנס לקטע קריטי

turn- תור של מי

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.

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

מימוש מנעול מכיל קטע קריטי:
קרא מנעול
אם מנעול פנוי, אז
כתוב שהמנעול תפוס.

דרכים לממש מנעולים

- 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, מסתמכת על מגנוני חומרה.
- ידוע כאלגוריתם המאפיה (bakery).[Lamport, 1978]
- שימוש במספרים:
  - חוט נכנס לוקח מספר.
  - חוט ממתין שמספרו הקטן ביותר נכנס לקטע הקריטי.

תיאום בין תהליכים:שיטות מתקדמות

מימוש מנעולים: חסימת פסיקות

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

- חסימת פסיקות מונעת החלפת חוטים ומבטיחה פעולה אטומית על המנעול
- למה מאפשרים פסיקות בתוך הלולאה?
- 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

test&set(boolvar)

– כתוב true ל- **boolvar** והחזר ערך קודם

מנעול תפוס – L = true; מנעול פנוי – L = false

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

המתנה בזבזנית

- spinlock מנעול שיש בו busy waiting:
  - בדוק האם המנעול תפוס (על-ידי גישה למשתנה).
  - אם המנעול תפוס, בדוק שנית.
- 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 המחכים

compare&swap(mem, R_1, R_2)

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

הנזק אינו רק שחוט הממתין לא עושה עבודה משמעותית אלא גם שהחוט בעל המנעול לא מתקרב לשחרור המנעול.

busy wait הוא לא תמיד דבר רע – התקורה על כניסה להמתנה היא 0 ולכן בהמתנות קצרות משתלם לא להוציא את החוט / תהליך לתור המתנה. לכן, משתמשים ב-spin locks ב-Linux במערכות מרובות מעבדים (ששם אין בעית priority inversion) לצורך פעולות המתנה קצרות

מנגנוני תיאום גבוהים יותר

- לנהל **תור** של החוטים הממתינים.
- נמצא במנגנוני תיאום עיליים: סמפורים,משתני תנאי,מוניטורים

סמפור- שני שדות:

- ערך שלם
- תור של חוטים / תהליכים ממתינים

wait(semaphore)

- מקטין את ערך המונה ב-1
- ממתינים עד שערכו של הסמפור אינו שלילי

signal(semaphore)

- מגדיל את ערך המונה ב-1
- משחרר את אחד הממתינים.

סמפור **OKToBuyMilk**, בהתחלה 1.

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

הסמפור תפוס,⇐ אם ערך הסמפור ≥ 0

וערכו (בערך מוחלט) הוא מספר הממתינים

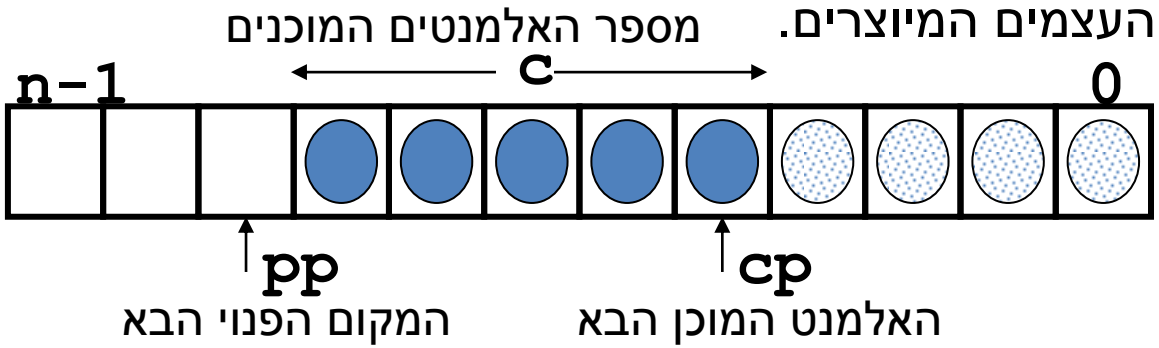
- סמפור **בינארי**
  - ערך התחלתי =1; זה גם הערך המקסימאלי.
  - מאפשר גישה בלעדית למשאב (בדומה למנעול).
- סמפור **מונה**
  - ערך התחלתי N>0.
  - שולט על משאב עם N עותקים זהים.
  - חוט יכול לעבור ()wait בסמפור כל עוד יש עותק פנוי של המשאב.

דוגמה: בעיית יצרן / 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.



מספר המקומות הפנויים

מספר האיברים המוכנים

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

דוגמה: 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	✗	✗

מונה מספר הקוראים
מגן על מונה מספר הקוראים

מניעה הדדית בין קוראים לבין כותבים
(ובין כותבים לעצמם)

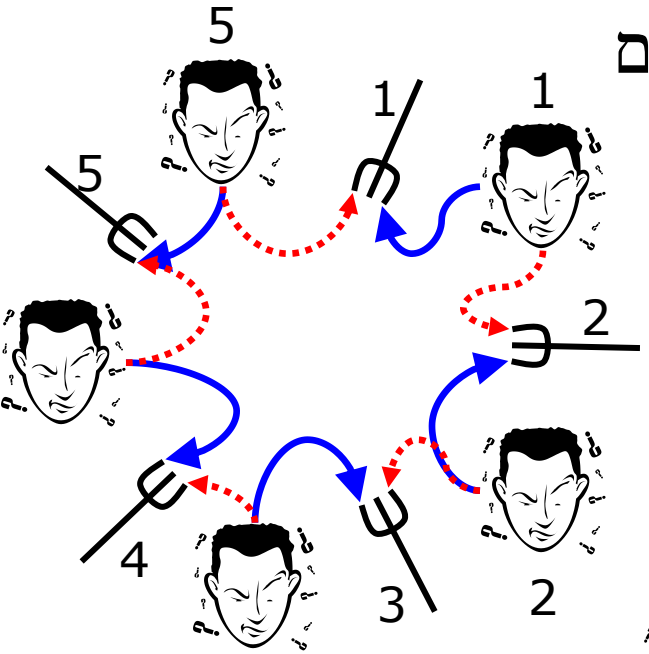
```
int r = 0;
semaphore sRead,
   initially 1
semaphore sWrite,
   initially 1
```

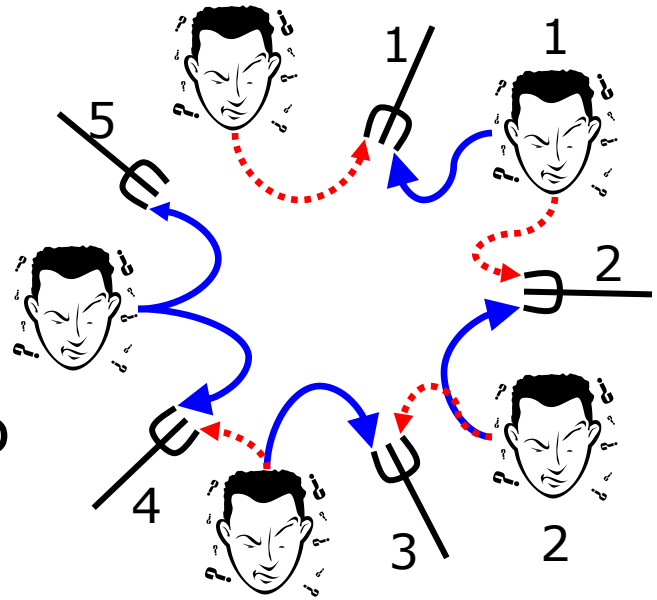
```
Reader:
   wait(sRead)
   r:=r+1
   if r=1 then
       wait(sWrite)
   signal( sRead)
   [Read]
   wait( sRead)
   r:=r-1
   if r=0 then
       signal(sWrite)
   signal( sRead)
```

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

דוגמא: הפילוסופים הסועדים

- כל פילוסוף מעביר את חייו ב-
  - חשיבה (thinking)
  - ניסיון להשיג מזלגות (hungry)
  - אוכל (eating)





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])
```

פיתרון פשוט

סמפור 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 שמשמאלו ⇦ קיפאון

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)
2. החזק והמתן
   - תהליך מחזיק משאב ומחכה למשאב אחר שבשימוש אצל תהליך אחר
3. לא ניתן להפקיע משאבים
4. המתנה מעגלית...
   - תהליכים $P_0, P_1, \ldots, P_{n-1}$
   - P_i מחכה למשאב המוחזק ע"י $P_{(i+1) \bmod n}$

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.

אם המעבד מגלה שהיו פסיקות לאחר ביצוע פעולה...

- שומר על המחסנית את כתובת החזרה,בד"כ כתובת ההוראה הבאה.
- אולי מידע נוסף:רגיסטרים מיוחדים, מידע נוסף, 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 שגרת הטיפול בפסיקה?

- לכל פסיקה שגרת טיפול משלה, הפועלת בהתאם לסוג הפסיקה
- 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.

- 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.

טיפול בפסיקות במערכת מרובת-מעבדים

- 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.

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.

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.

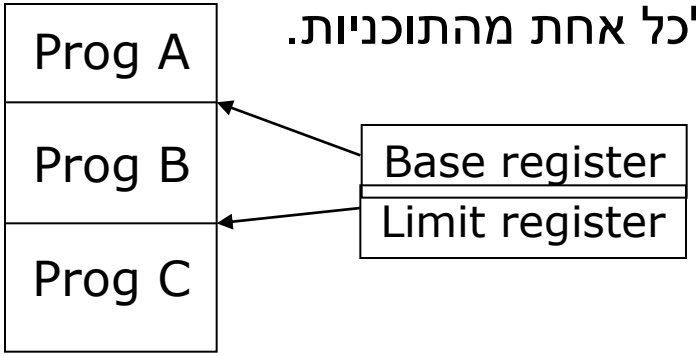
- מערכת ההפעלה מחלקת משאבים מצומצמים בין כמה תהליכים.
- דרישות הזיכרון (פיזי) של תהליך לא על-חשבון תהליך אחר.

שיטה ישנה: חלוקה קבועה

- מערכת ההפעלה מחלקת את הזיכרון הפיזי לחתיכות בגודל קבוע size שיכולות להכיל חלקים ממרחב הכתובות של תהליך.לכל תהליך מוקצית חתיכת זיכרון פיזי.
  - מספר התהליכים הרצים ≥ מספר החתיכות.

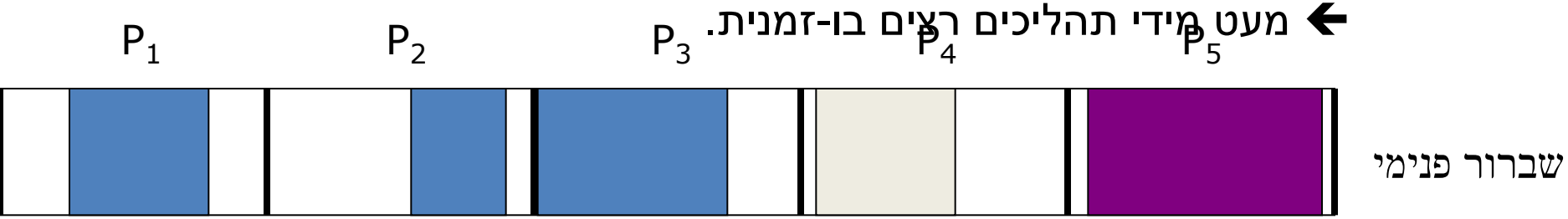
הגנה על הזיכרון

- 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.



בעיות 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.



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?

external fragmentation שיברור חיצוני שאריות מקום בין החתיכות שלא מתאימות לכלום.

זיכרון וירטואלי

- 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).
- 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?

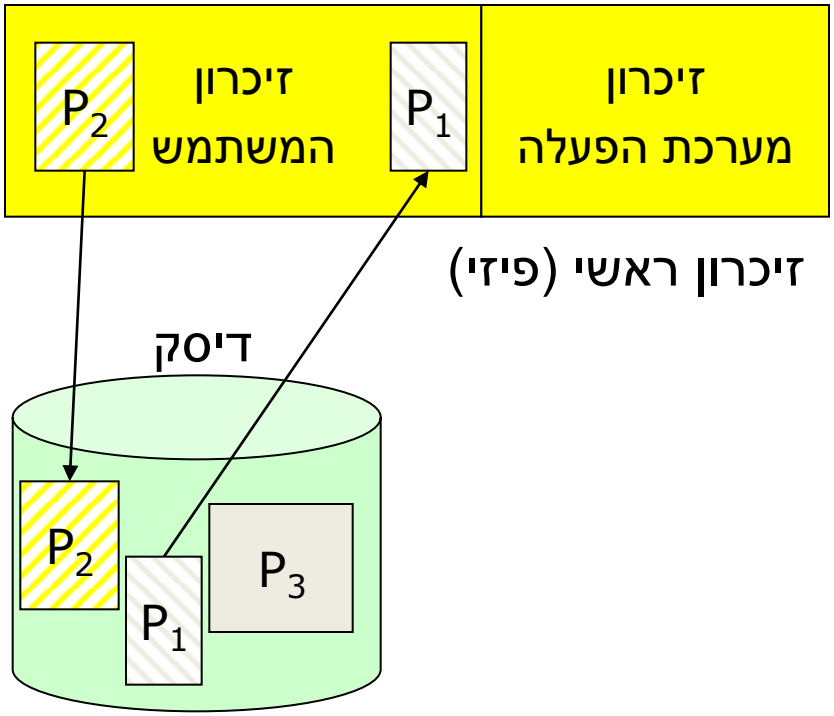
... בכלל משתמש בו?

- השטח הכולל שבשימוש קטן בהשוואה למרחב הזיכרון כולו

← מקצים זיכרון רק אם משתמשים בו.

<u>עקרון הלוקליות:</u> תהליך ניגש רק לחלק מזערי של הזיכרון שברשותו בכל פרק זמן נתון

Swapping



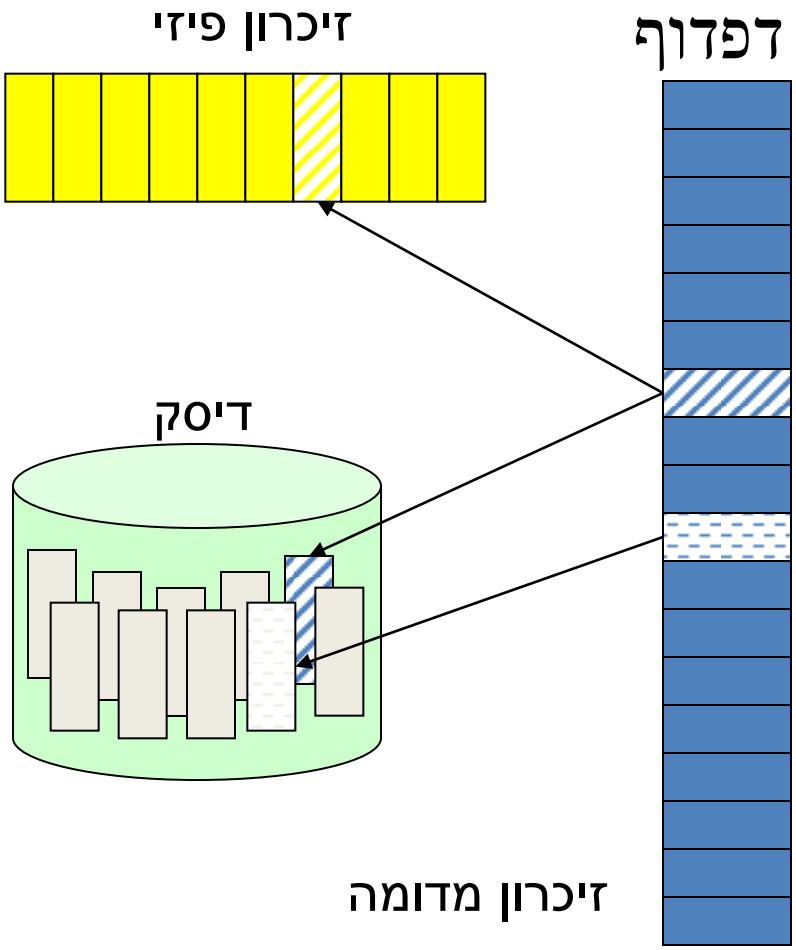
- זיכרון של תהליך שאינו רץ מועבר לדיסק (swap-out).
- כאשר תהליך חוזר לרוץ, מקצים לו מחדש מקום ומביאים את הזיכרון שלו (swap-in).

דורש מנגנון תמיכה...

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.

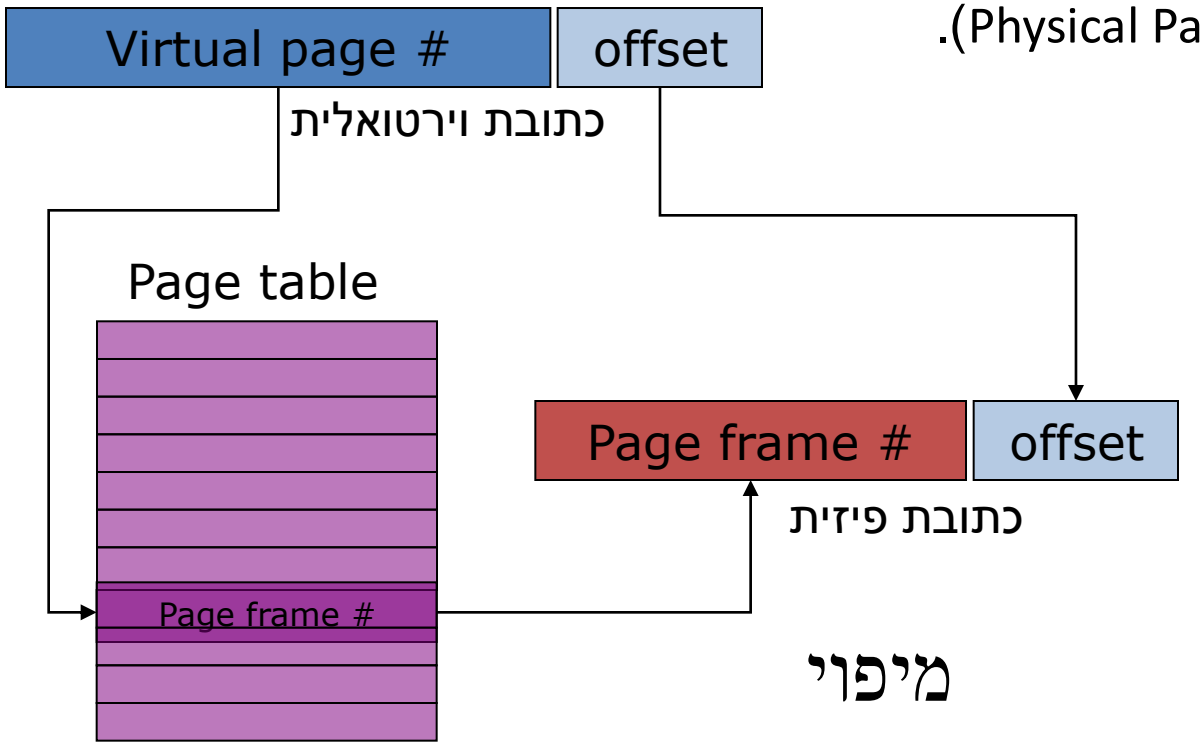
שיטה מודרנית: דפדוף



- 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
- כל מסגרת בזיכרון הפיזי יכולה להחזיק כל דף וירטואלי.
- כל דף וירטואלי נמצא בדיסק.
- חלק מהדפים הוירטואליים נמצאים בזיכרון הפיזי.
- 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).
- ה VPN מהווה מצביע לטבלת הדפים ומאפשר למצוא את מספר המסגרת שבו נמצא הדף (Physical Page Number).



הוספת ה offset נותנת את הכתובת עצמה.

מיפוי

דוגמא

- 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.

כתובת וירטואלית | 00000000011100000000**110000000000**

תתורגם ל | 00000000100100000000**110000000000**

כניסות בטבלת הדפים

V	R	M	protect	Page frame #

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. PF נקראת גם "פסיקת דף" או "חריגת דף"
  - 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 מ: גישה לא חוקית לזיכרון, גישה לדף לא מוקצה ועוד

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.

הרע

- 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).

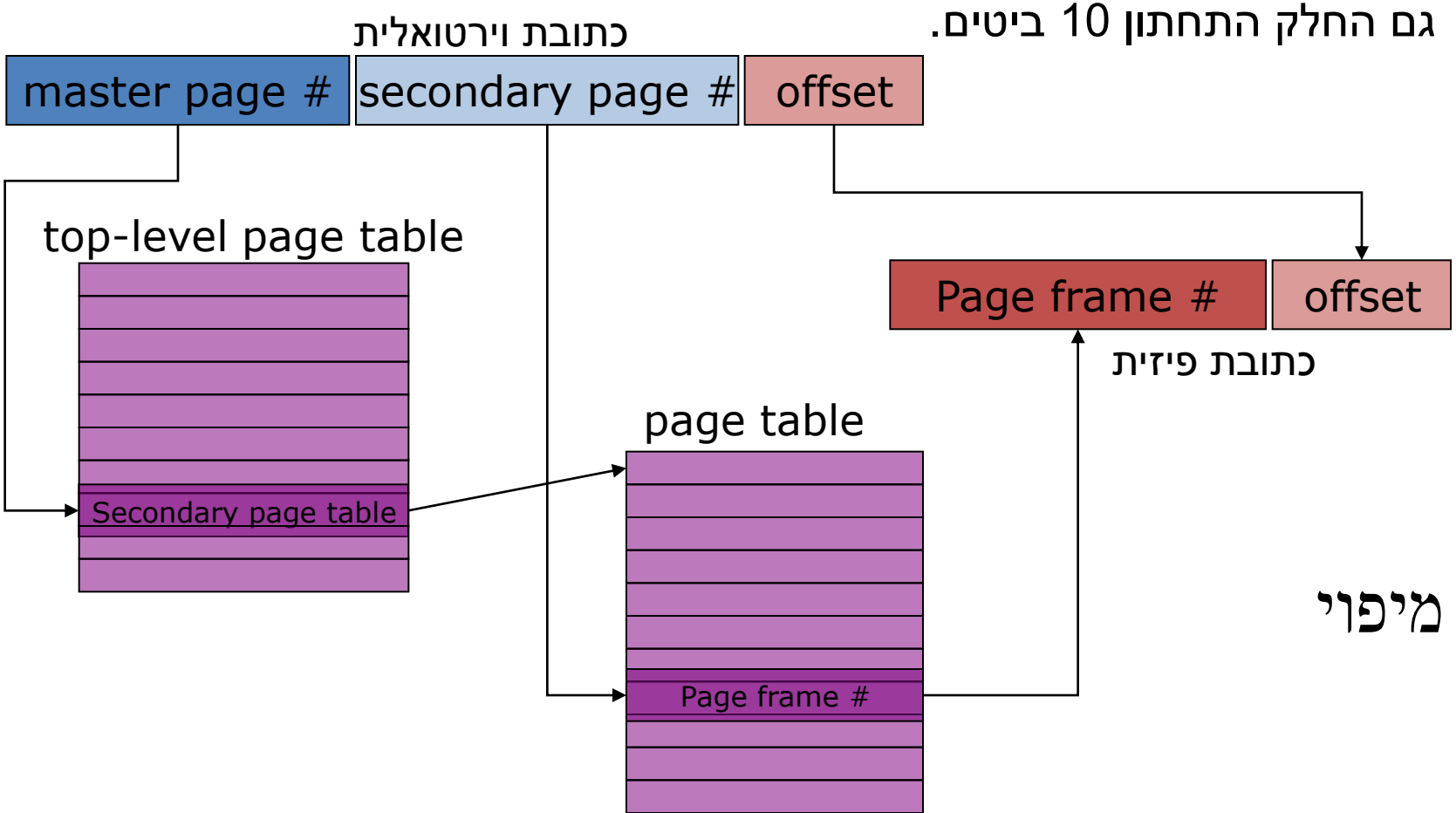
חלוקת הכתובת בשתי-רמות

Virtual page #	offset

master page #	secondary page #	offset

לכתובת וירטואלית יש שלושה חלקים:

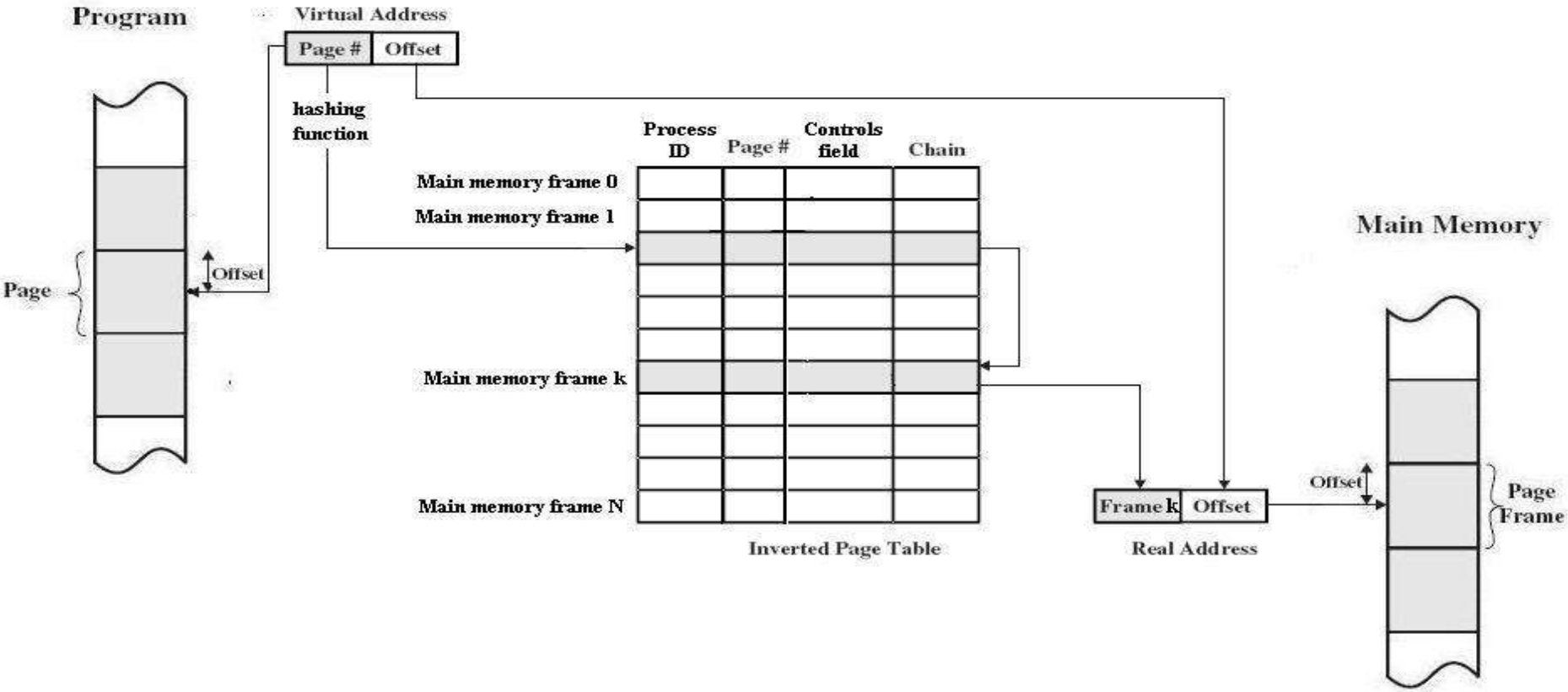
- רצוי שהטבלה ברמה העליונה תכנס בדף אחד
  - 4KB = 1024 כניסות כל-אחת עם 4 בתים.
  - החלק העליון של הכתובת צריך להיות עם 10 ביטים.
  - גם החלק התחתון 10 ביטים.



מיפוי

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 של 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...

Translation Lookaside Buffer (TLB)

- מטמון חומרה אשר שומר מיפויים 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.

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+Δ.)

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 מפנה את הדף שנטען לפני הכי הרבה זמן

- האלגוריתם האופטימאלי

- אם כל הגישות לזיכרון ידועות מראש...
- האלגוריתם **החמדן** מפנה מהזיכרון את הדף שהזמן עד לגישה הבאה אליו הוא הארוך ביותר.

- Least Recently Used (LRU)

מפנה את הדף שהגישה האחרונה אליו היא המוקדמת ביותר

מימוש LRU

- 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 ההזדמנות השנייה

reference bit לכל דף,בגישה לדף, הביט מודלק.

- בפינוי דפים, מערכת ההפעלה עוברת באופן מחזורי על כל הדפים
  - אם reference bit = 0, הדף מפונה.-- אם reference bit = 1, מאפסים את הביט.

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.

סגמנטים

- 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.

- 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
- Linux מכילה מנגנון סגמנטציה פנימי בצורה של **איזורי זיכרון**
  - דפדוף של כל הסגמנטים עם טבלת דפים עם **שלוש-רמות**.

מערכת קבצים

- קבצים מאפשרים שמירת מידע לטווח בינוני וארוך.למרות הפעלות חוזרות של תוכנית, איתחולים, ונפילות.

בניגוד לזיכרון הראשי, זיכרון משניאינו מאפשר ביצוע ישיר של פקודות או גישה לבתים.

שומר על מידע לטווח ארוך (לא-נדיף, non-volatile).

גדול, זול, ואיטי יותר מזיכרון ראשי. **מטרות:**

- הפשטה מתכונות ספציפיות של ההתקן הפיזי גישה דומה לדיסק, CD-ROM
- זמן ביצוע סביר.
- ארגון נוח של הנתונים.
- הפרדה בין משתמשים שונים (protection).
- הגנה (security).

מבנה לוגי של מערכת קבצים

- אוסף של קבצים.
- קובץ הוא מידע עם תכונות שמנוהלות על-ידי המערכת.
  - מכיל מספר בתים/שורות/רשומות בגודל קבוע/משתנה.
  - קובץ יכול להיות מסוג מסוים
    - חלק מהסוגים מזוהים על-ידי המערכת: מדריך, link, mount
    - ...או על-ידי אפליקציות: .exe, .doc, .html, .jpg.
    - במקביל, ניתן להתייחס לתכולת הקובץ: בינארי או טקסט

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.

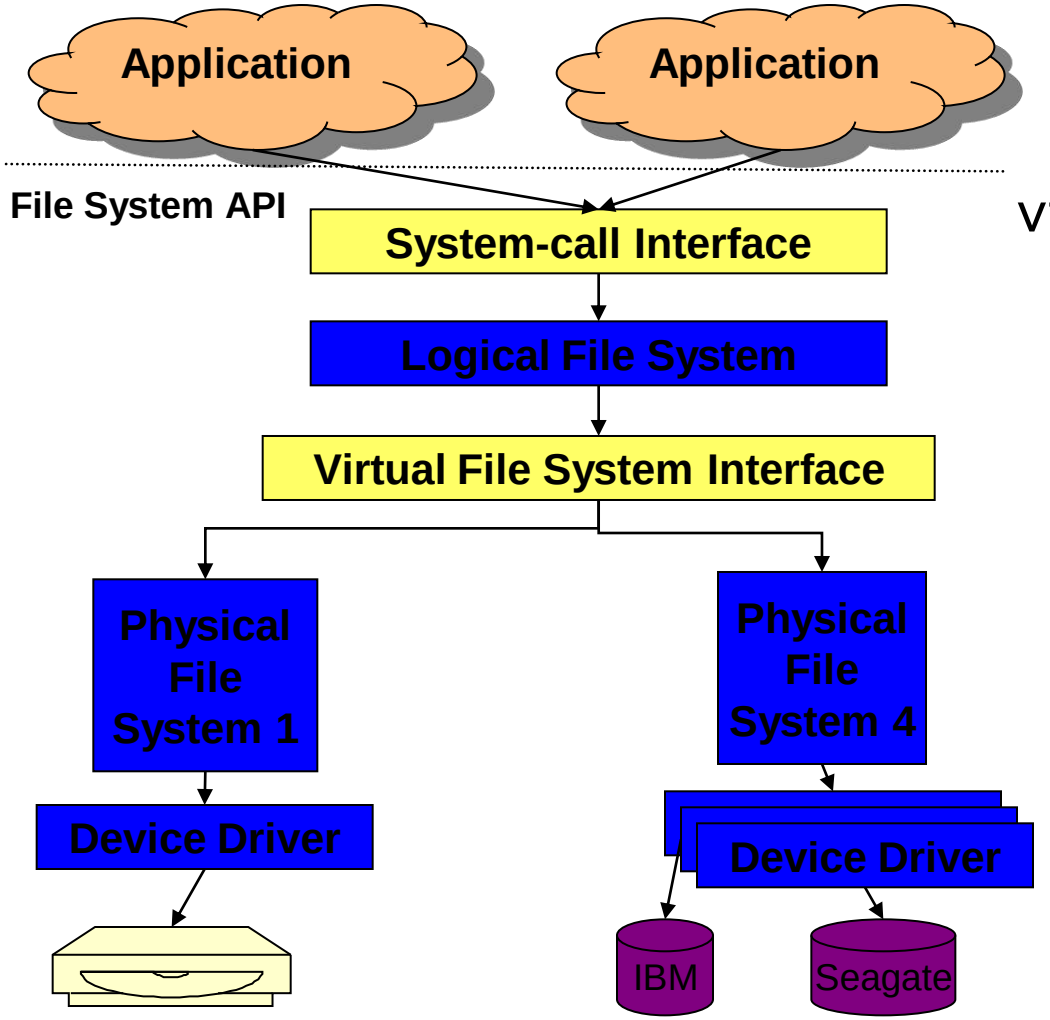
ארגון מערכת קבצים

- **מחיצות** (partitions), בד"כ לפי התקנים (devices).
- **מדריכים** (directories), טבלאות הקבצים בתוך המחיצה.

מדריכים

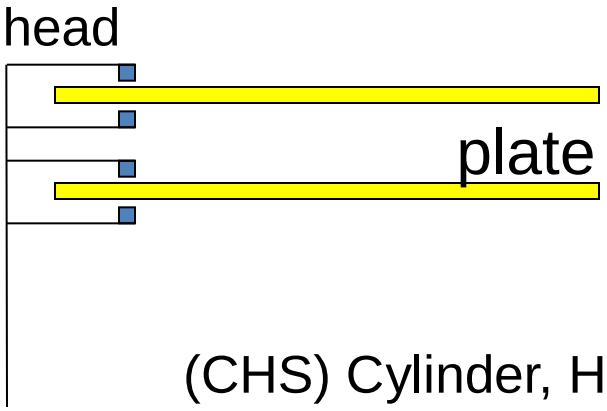
- המדריך הוא מבנה נתונים מופשט, אשר שומר את התכונות של כל קובץ הנמצא בו.
- תומך בפעולות הבאות:
  - מציאת קובץ (לפי שם)
  - יצירת כניסה
  - מחיקת כניסה
  - קבלת רשימת הקבצים בתוך המדריך
  - החלפת שם של קובץ

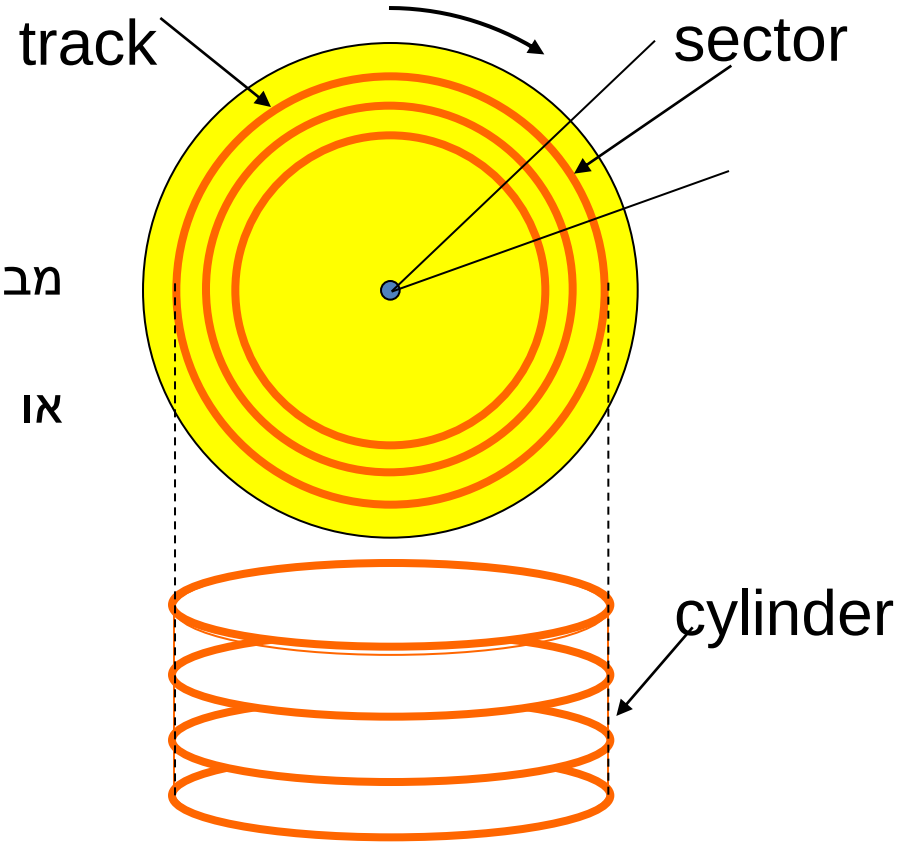
מבנה מערכת קבצים טיפוסית



- logical file system
- virtual file system interface
  - מנשק אחיד לכל מערכות הקבצים הספציפיות.
  - למשל, vfs_read, vfs_write, vfs_seek
- physical file system
  - מימוש מנשק ה-VFS עבור מערכת קבצים ספציפית
    - למשל, מעל דיסק, דיסקט, RAM, CD, *רשת* וכו'
  - מתכנן איך לפזר את הבלוקים.
  - בהמשך, נתרכז בו.
- device drivers
  - קוד שיודע איך לפנות להתקן ספציפי
  - דיסקים (לפי מודל), DVD, וכדומה.
  - מתחיל את הפעולה הפיזית, ומטפל בסיומה.
  - מתזמן את הגישות, על מנת לשפר ביצועים.

מבנה הדיסק





מבנה כתובת

(CHS) Cylinder, Head, Sector

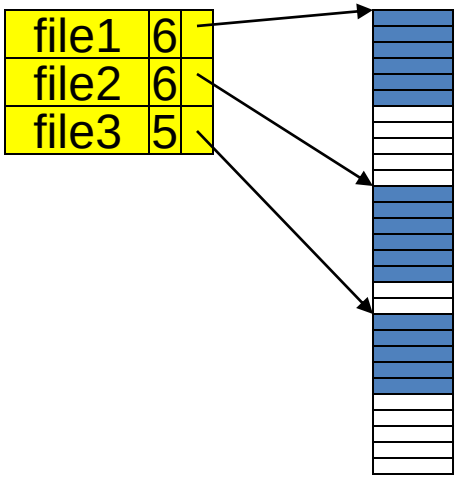
או

(LBA) Logical Block Array

מדדים

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

מיפוי קבצים: הקצאה רציפה

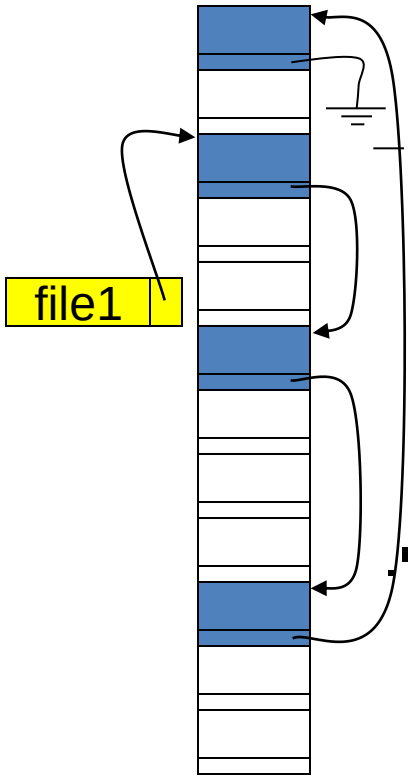


- בבלוקים באורך קבוע
  - כפולה של גודל סקטור (נע בין 512B – 4KB)
- המשתמש מצהיר על גודל הקובץ עם יצירתו.
- מחפשים בלוקים רצופים שיכולים להכיל את הקובץ.
- הכניסה במדריך מצביעה לבלוק הראשון בקובץ.

✓ גישה מהירה (סדרתית וישירה)

✗ שיברור פנימי וחיצוניף קשה להגדיל קובץ

הקצאה משורשרת



- כל בלוק מצביע לבלוק הבא.
- הכניסה במדריך מצביעה לבלוק הראשון בקובץ

✓ 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.

אמינות

- 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.

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 בכתיבה אחת.

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 תוויים 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 תוויים.

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).

שיטת 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.

מושגי יסוד

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- 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 Sarved -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.

Disk Reliability

הדיסק הוא יחידה עם מנגנונים מכניים, ועם חלקים נעים,ייתכנ 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.

<u>יתרון:</u>

הרבה דיסקים קטנים וזולים במקום דיסק אחד גדול ויקר.

<u>חסרון:</u>

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

<u>RAID</u>

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.

- <u>RAID level 0</u>– מערך של דיסקים (disk stripping).
- <u>RAID level 1</u>– נקרא גם mirroring אן shadowing. תמיד יהיו 2 עותקים מכל דבר. כל מידע נכתב גם לעותק. במקרה של נפילה תמיד ניתן לשחר מהעותק.

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