מערכות הפעלה

6

זיכרון

ריבוי תהליכים multiprogramming

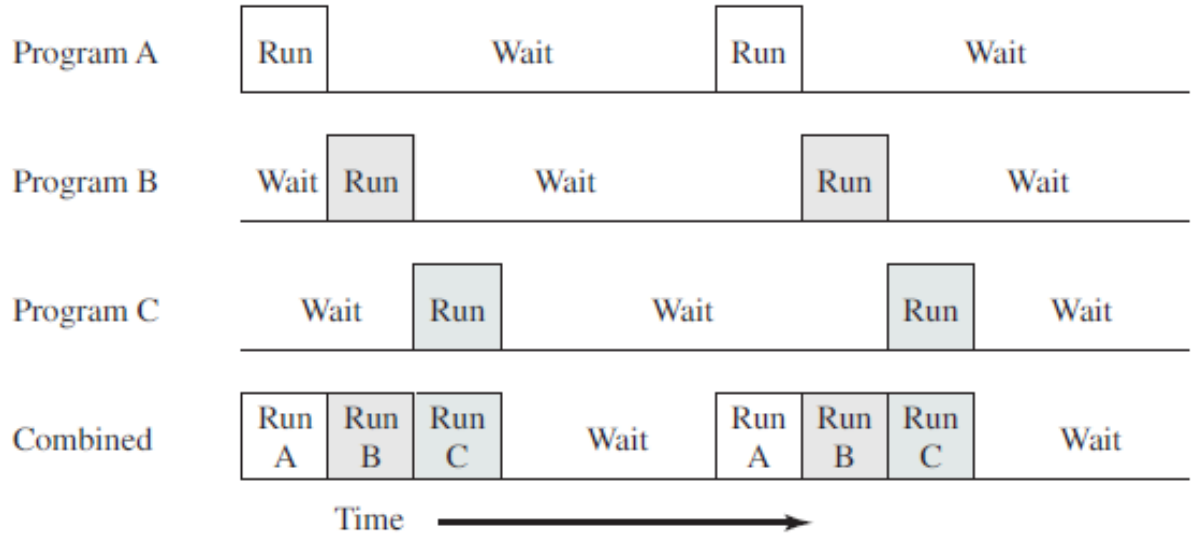
המטרות של ריבוי תהליכים:

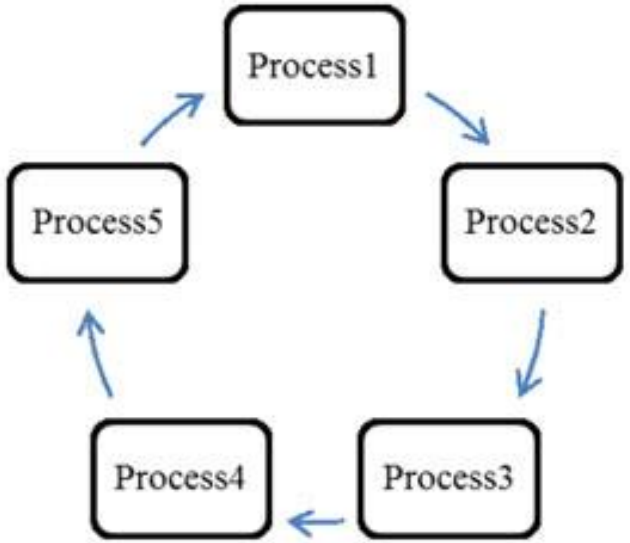
1. ניצול המעבד

- התקני קלט/פלט הם איטיים יחסית לפעולת המעבד, אם תתבצע תכנית אחת בלבד, במשך רוב הזמן המעבד לא יהיה מנוצל.
- לפיכך מערכת ההפעלה מריצה כמה תכניות ובזמן שתכנית ממתינה לסיום קלט/פלט (או למאורע אחר), תכנית אחרת יכולה לרוץ.

2. תגובה מהירה

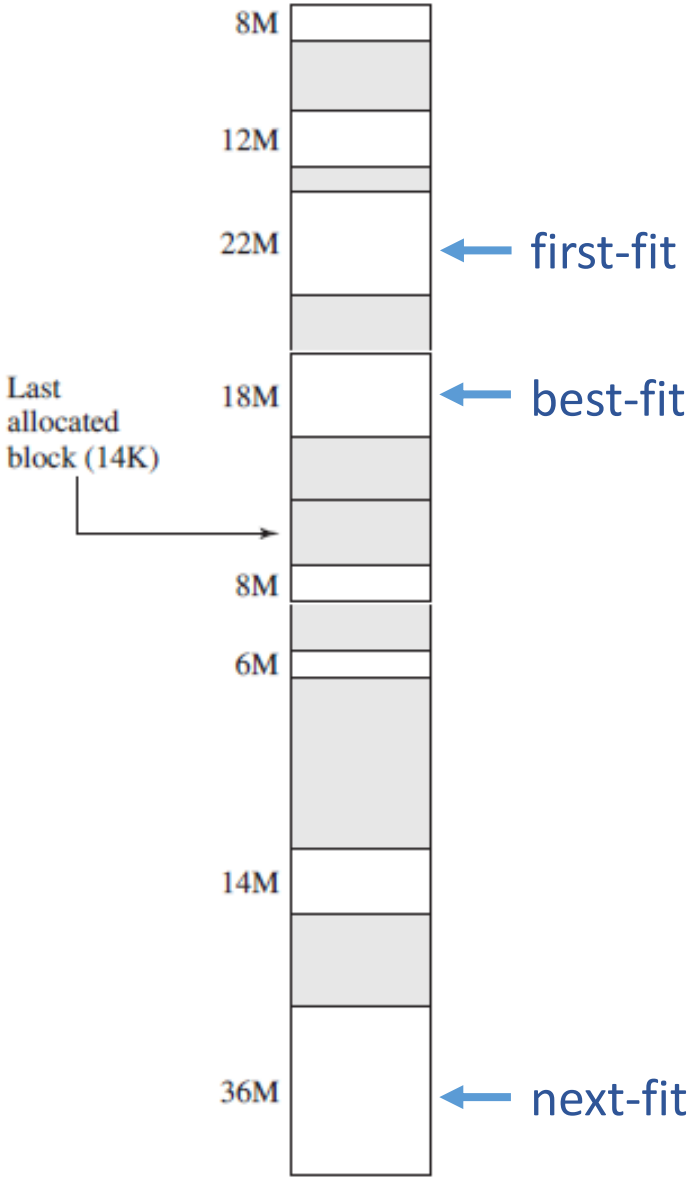
- מערכת ההפעלה מפסיקה ריצה של תכנית ונותנת את המעבד לתכנית אחרת לפי פסיקת שעון.
- המטרה היא לאפשר זמן תגובה מהיר למשתמשים ולתהליכים.





הקצאת זיכרון לפי גודל התהליך

הקצאה עבור תהליך בגודל 16MB

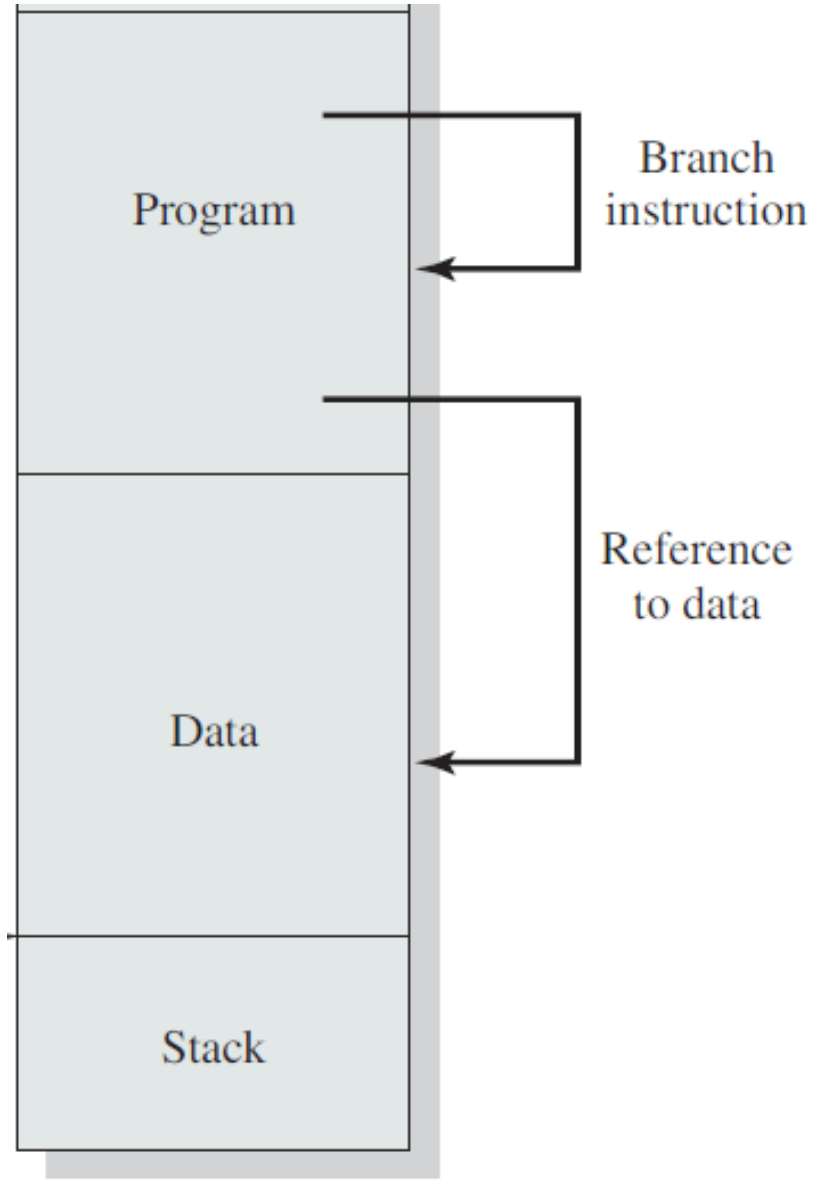


- כדי להריץ תהליכים, יש לטעון אותן לזיכרון.
- חלוקת הזיכרון למחיצות בגודל קבוע אינה יעילה בגלל **שבירה פנימית**.
  - עדיף לחלק למחיצות שגודלם משתנה לפי גודל התהליך.
- לאחר הקצאות ושחרורים ייווצרו מחיצות פנויות, באיזו מהן נבחר?

- first-fit - begins to scan memory from the beginning and chooses the first available block that is large enough.
- next-fit - begins to scan memory from the location of the last placement, and chooses the next available block that is large enough.
- best-fit - chooses the block that is closest in size to the request.

- בהקצאה למחיצות בגודל משתנה תהיה **שבירה חיצונית**.
  - פתרון אפשרי הוא לדחוס את המחיצות לכיוון התחלת הזיכרון, פתרון אחר נראה להלן.

הזזה - Relocation



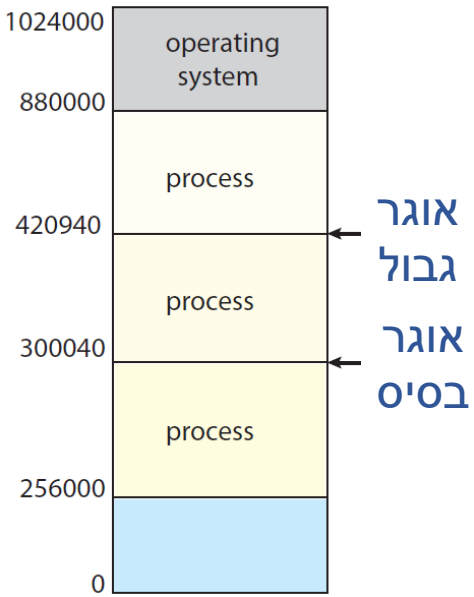
בעיה בחלוקת הזיכרון לכמה תהליכים:

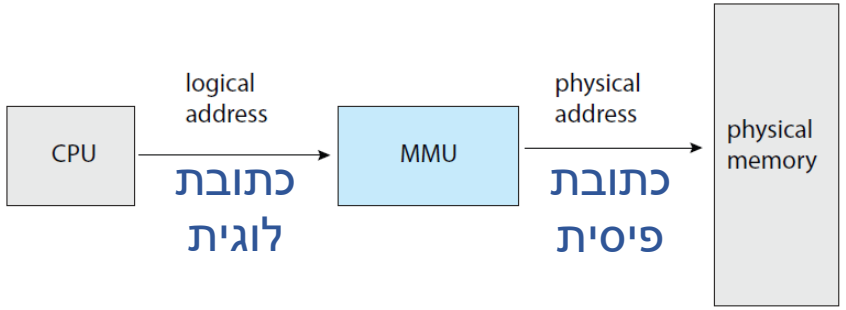
- קוד הריצה של תכנית (לאחר קומפילציה) כתוב יחסית לכתובת 0:
  - קוד הריצה מכיל פקודות שקוראות (LOAD) נתונים וכותבות (STORE) נתונים לכתובות בזיכרון יחסית לכתובת 0.
  - קוד הריצה מכיל פקודות שמבצעות קפיצה (JUMP) לכתובות בזיכרון יחסית לכתובת 0.
  - מאחר שישנם כמה תהליכים בזיכרון, תהליך נטען לכתובת שאינה 0 ואינה ידועה מראש.

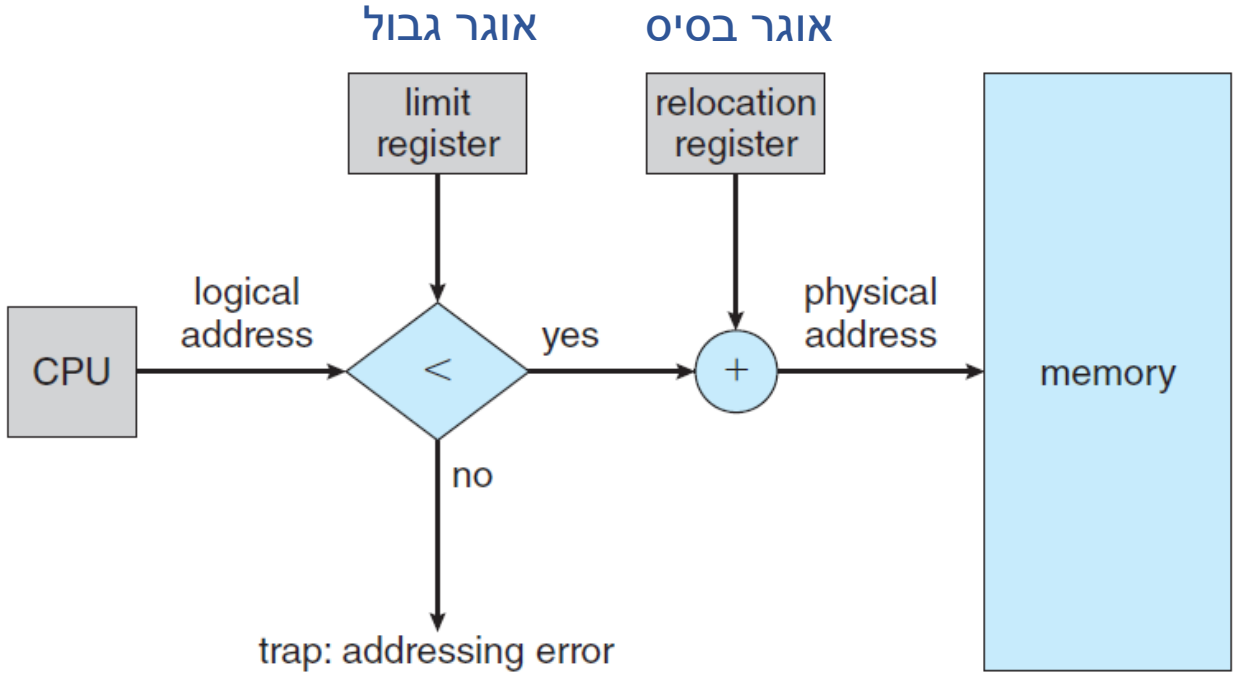
פתרון:

- בעבר, בזמן טעינת התכנית היו עורכים את קוד הריצה ומשנים את הכתובות בהתאם למקום אליו נטענת התכנית.
- כיום משתמשים בחומרה כדי לתרגם את הכתובות הלוגיות של תכנית לכתובות בזיכרון הפיסי.

חומרה להזזת כתובות



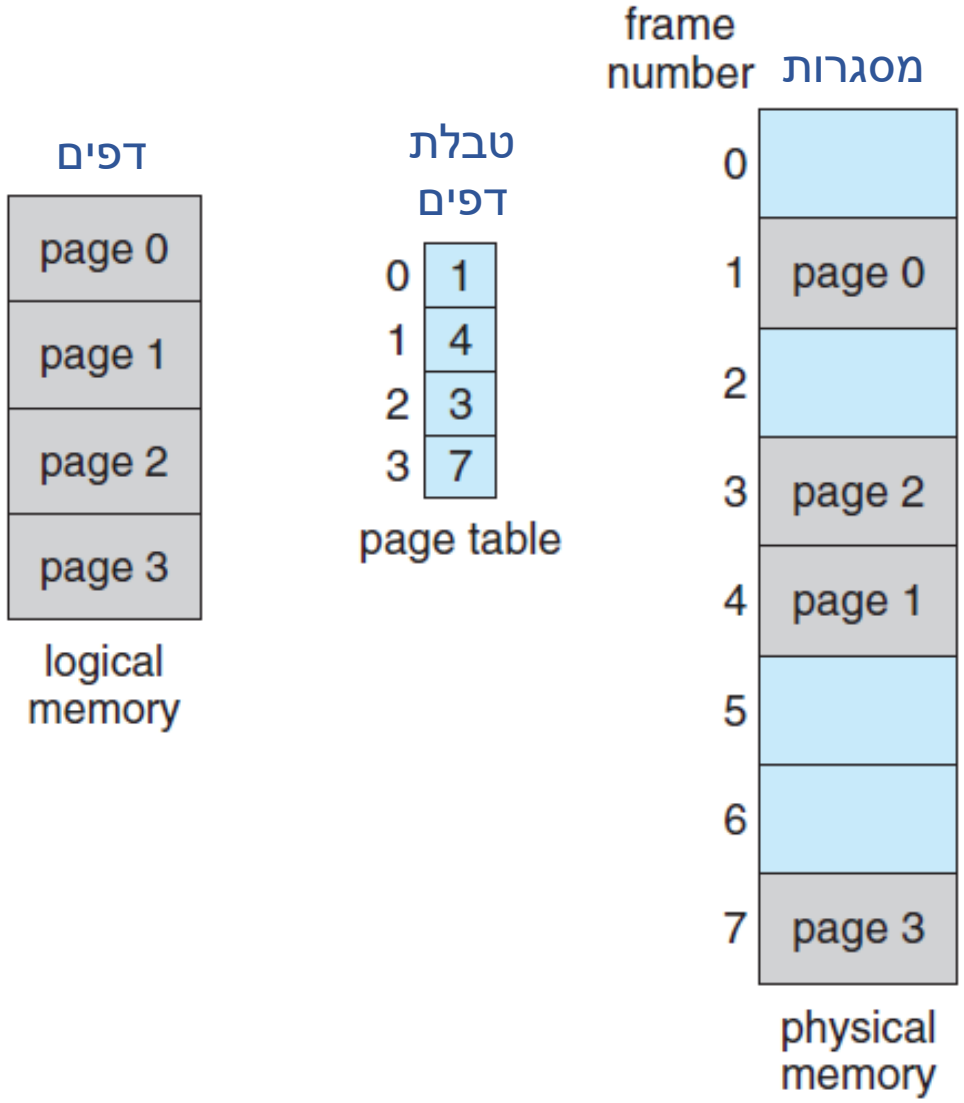




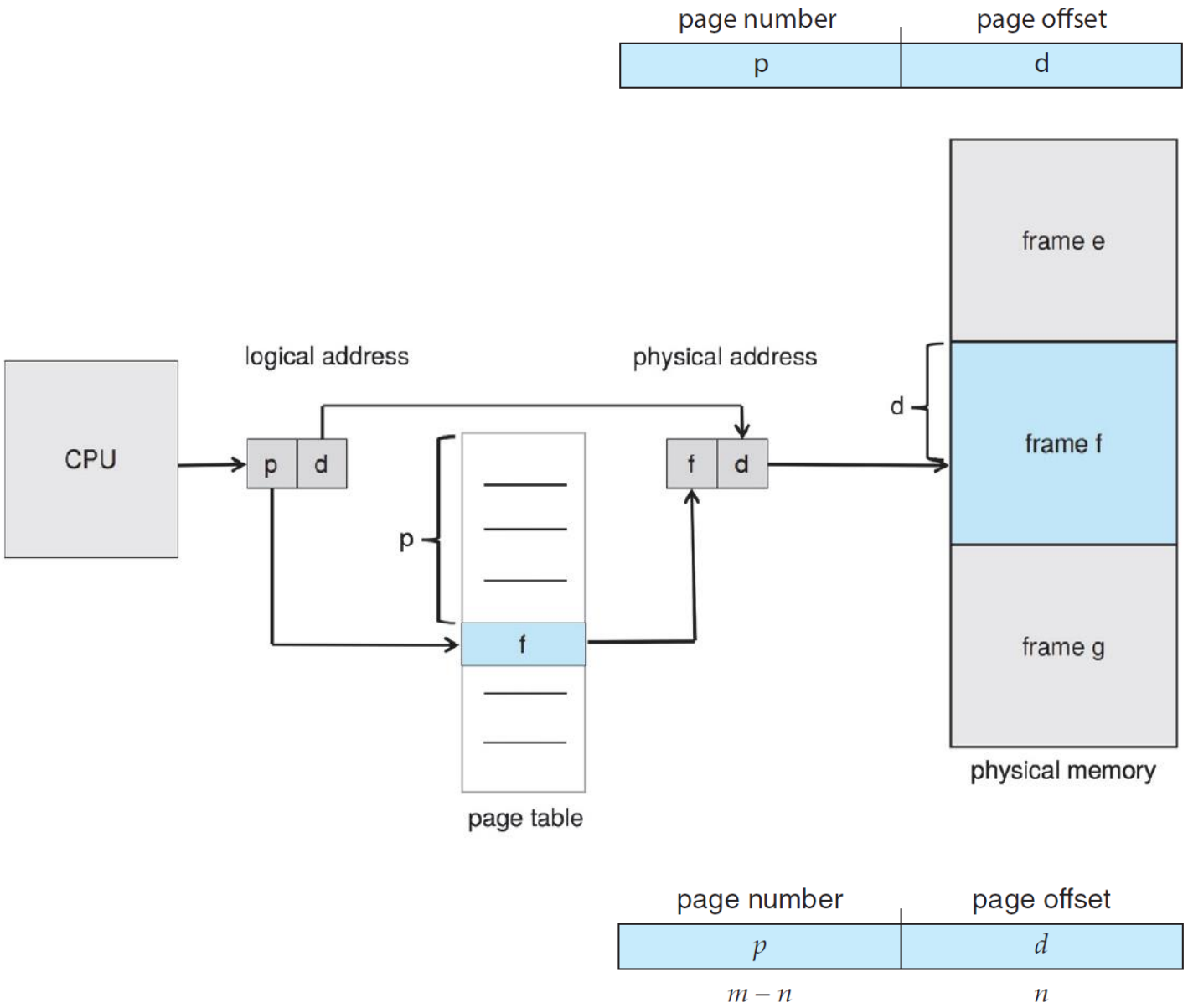
- רכיב החומרה לניהול הגישות לזיכרון (MMU), נמצא בין המעבד לזיכרון.
- הרכיב מכיל שני אוגרים: אוגר בסיס ואוגר גבול.
  - כאשר מערכת ההפעלה טוענת תכנית לזיכרון, היא מכניסה לאוגר הבסיס את כתובת תחילת המחיצה שנבחרה, ומכניסה לאוגר הגבול את כתובת סוף התכנית.
  - לכל כתובת לוגית שהמעבד מוציא, רכיב החומרה מוסיף את אוגר הבסיס, ויוצר את הכתובת בזיכרון הפיסי.
  - רכיב החומרה גם בודק שהכתובת לא גבוהה מאוגר הגבול, אחרת הוא יוצר פסיקה.
- תהליך לא יוכל לגשת לזיכרון של תהליך אחר או של מערכת ההפעלה.

חלוקה לדפים - Paging

- כדי לפתור את בעיית **השבירה החיצונית** מקצים את הזיכרון ביחידות של דפים:
  - הזיכרון **הלוגי** של כל תהליך מחולק לחלקים שווים המכונים **דפים** (גודל שכיח הוא 4K).
  - הזיכרון **הפיסי** (RAM) מחולק לחלקים בגודל שווה (באותו גודל של הדפים) המכונים **מסגרות**.
- **מערכת ההפעלה** מכינה **טבלת דפים** שמכילה מיפוי של הדפים הלוגיים של התהליך למסגרות פנויות בזיכרון הפיסי.
  - כשהמעבד מוציא כתובת לוגית לזיכרון, **החומרה** (MMU) משתמשת בטבלת הדפים כדי לתרגם את הכתובת הלוגית לפיסית.
- אין שבירה חיצונית, אבל יש **שבירה פנימית** של הדף האחרון.



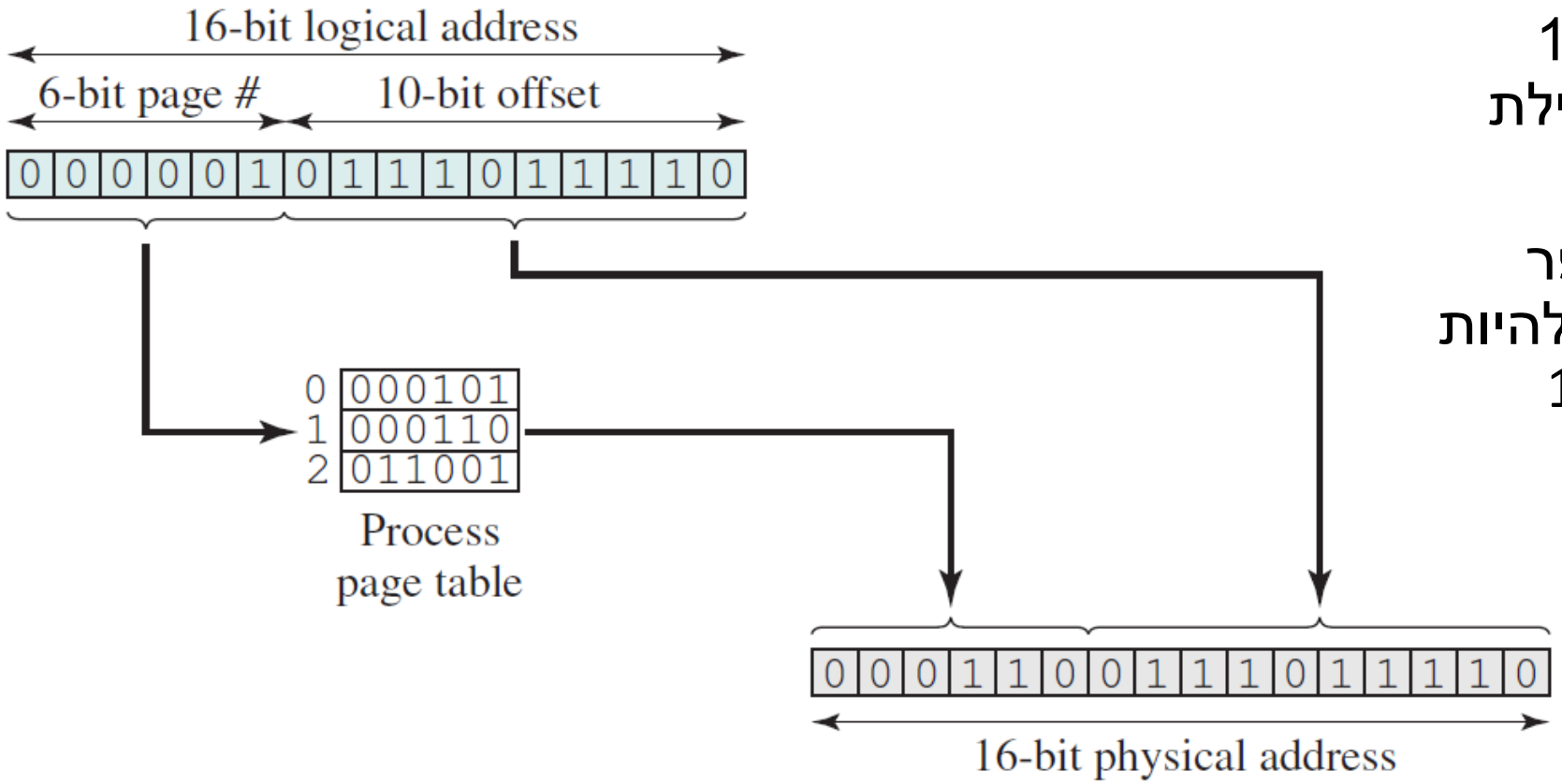
תרגום כתובת לוגית לפיסית בחלוקה לדפים



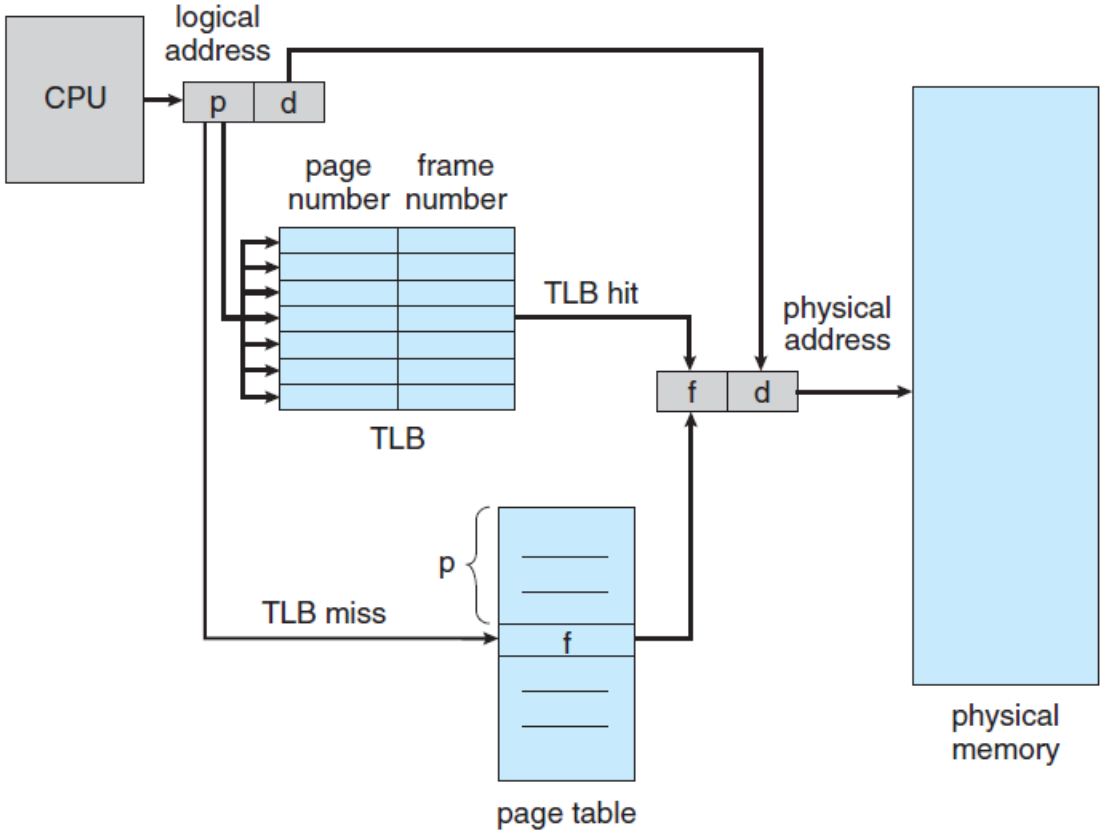
- כתובת לוגית מחולקת לשני חלקים: מספר הדף ומרחק הבית מתחילת הדף (offset).
- מספר הדף משמש **כאינדקס** בטבלת הדפים שיש **לכל תהליך**.
- טבלת הדפים מכילה עבור כל דף לוגי את כתובת התחלת הדף הפיסי.
- מרחק הבית (offset) בדף הפיסי הוא כמו המרחק בדף הלוגי.
- אם כן, הכתובת הפיסית מתקבלת מחיבור תחילת הדף הפיסי והמרחק.
- אם גודל הכתובת הוא 2^m וגודל הדף 2^n, אזי m-n הביטים הגבוהים יציינו את מספר הדף ו- n הביטים הנמוכים את המרחק:

דוגמה, תרגום כתובת לוגית לפיסית

- כתובת בגודל 16 ביטים.
- גודל דף 1K=1024 בתים.
- מאחר שגודל הדף הוא 2^{10}=1024, יש צורך ב- 10 ביטים עבור המרחק מתחילת הדף (offset).
- נשארו 6 ביטים עבור מספר הדף, אם כן תכנית יכולה להיות בגודל 2^6=64 דפים של 1K כלומר 64K לכל היותר.



מטמון לתרגום מהיר של כתובת - TLB



בעיה: גישה לזיכרון באמצעות טבלת דפים דורשת שתי גישות לזיכרון, אחת לטבלת הדפים ואחת לנתון.

פתרון: כדי שזמן הגישה לא יוכפל, משתמשים במטמון (TLB) שמכיל את הכניסות האחרונות של טבלת הדפים.

- המטמון בנוי מזיכרון מהיר והחיפוש בו נעשה במקביל.
- כל כניסה במטמון מכילה את מספר הדף הלוגי ואת מספר המסגרת הפיסית.
- כשהמעבד מוציא כתובת, ה- MMU מחפש תחילה במטמון, אם לא נמצא (TLB miss) ה- MMU מחפש בטבלת הדפים ומוסיף למטמון.
- אם המטמון מלא יש צורך להחליף כניסה.
- כאשר מחליפים תהליך צריך לרוקן את המטמון.

יחס הפגיעה במטמון – TLB hit ratio

- יחס הפגיעה במטמון TLB הוא מספר הפעמים באחוזים שה- MMU חיפש במטמון ומצא את הדף.
- נניח שזמן הגישה לזיכרון הוא 10 ננו-שניות, אזי אם הדף במטמון TLB אין תוספת לזמן הגישה, מאחר שמטמון TLB הוא מאוד מהיר.
- אם הדף אינו במטמון TLB, יש צורך בגישה נוספת לטבלת הדפים שנמצאת בזיכרון הרגיל ולכן זמן הגישה יהיה 20 ננו-שניות.
- נניח שיחס הפגיעה הוא 80%, אזי זמן הגישה האפקטיבי:

$$\text{effective access time} = 0.80 \times 10 + 0.20 \times 20 = 12 \text{ nanoseconds}$$

ההאטה שנגרמה בגלל דפדוף היא אם כן 20% (מ- 10 ל- 12).

- אם יחס הפגיעה הוא 99% (יותר מציאותי), אזי זמן הגישה האפקטיבי:

$$\text{effective access time} = 0.99 \times 10 + 0.01 \times 20 = 10.1 \text{ nanoseconds}$$

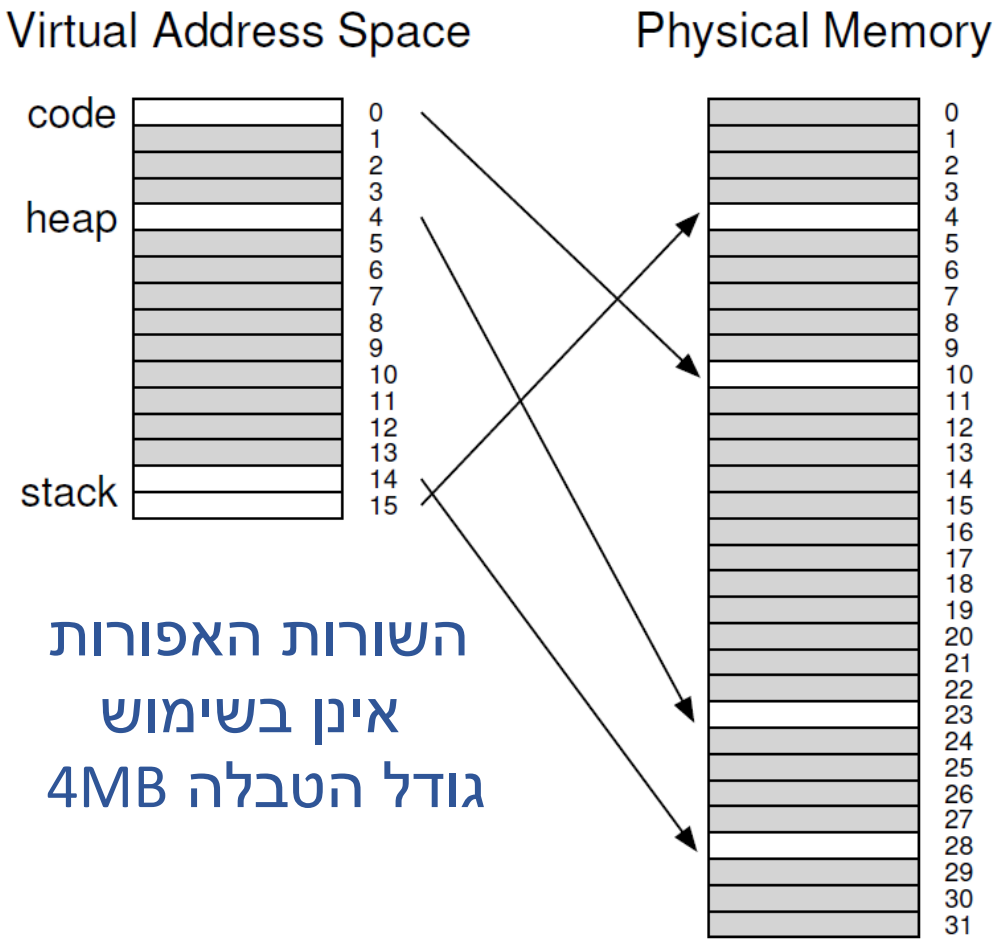
ההאטה שנגרמה בגלל דפדוף היא 1% (מ- 10 ל- 10.1).

בעיה: טבלת הדפים גדולה

- מרחב הכתובות הלוגיות של תהליך במחשב 32 ביט הוא 2^{32}=4GB, ובמחשב 64 ביט הוא 2^{64}.
- נניח מחשב 32 ביט וגודל דף 2^{12}=4K, אם כן טבלת הדפים תכיל כמיליון כניסות:

$$2^{32} / 2^{12} = 2^{20}$$

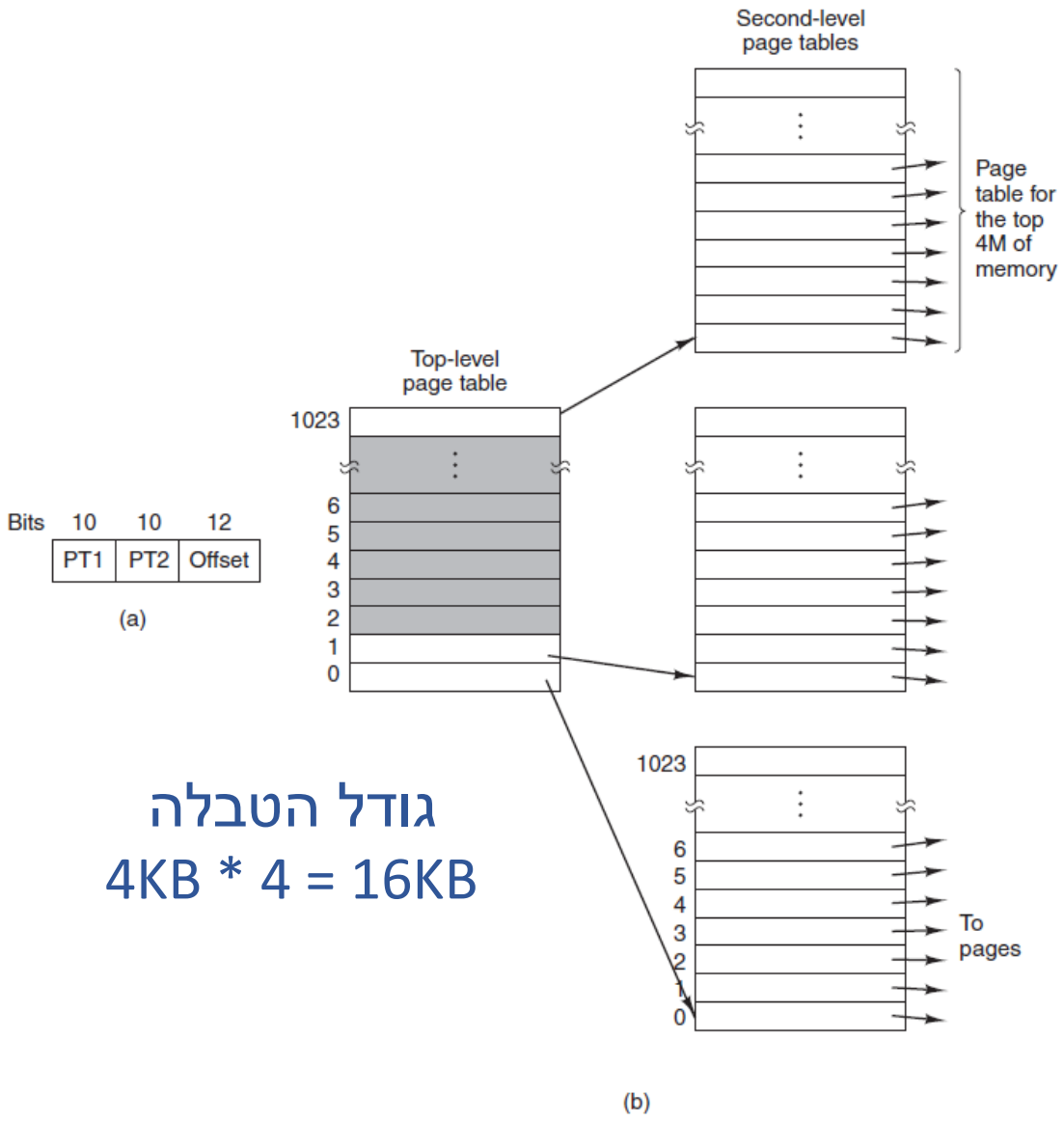
- אם כל כניסה בטבלת הדפים היא בגודל 4 בתים, הטבלה **עבור כל תהליך** תתפוס 4MB בזיכרון הפיסי!
- חלק גדול מהטבלה אינה בשימוש, אבל אם יש מצביעים לדפים בחלק התחתון ובחלק העליון יש צורך בכל הכניסות שביניהם.



השורות האפורות אינן בשימוש
גודל הטבלה 4MB

פתרון: מפת דפים היררכית

- אפשר להקטין את טבלת הדפים אם נבנה אותה בשתי רמות, כלומר טבלה של מצביעים לטבלאות דפים.
- הרמה השנייה תכיל רק את המצביעים שבשימוש.
- כאשר הדף לא נמצא במטמון TLB, אם יש שתי רמות, יש צורך בשתי גישות לזיכרון כדי לחפש בטבלת הדפים.
- כשמרחב הכתובות הוא 48 או 64 ביטים, משתמשים בשלוש או ארבע רמות.

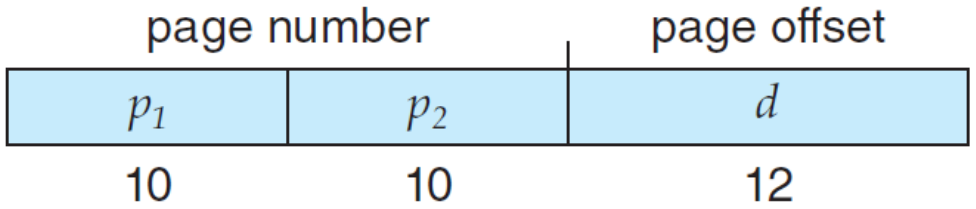


גודל הטבלה

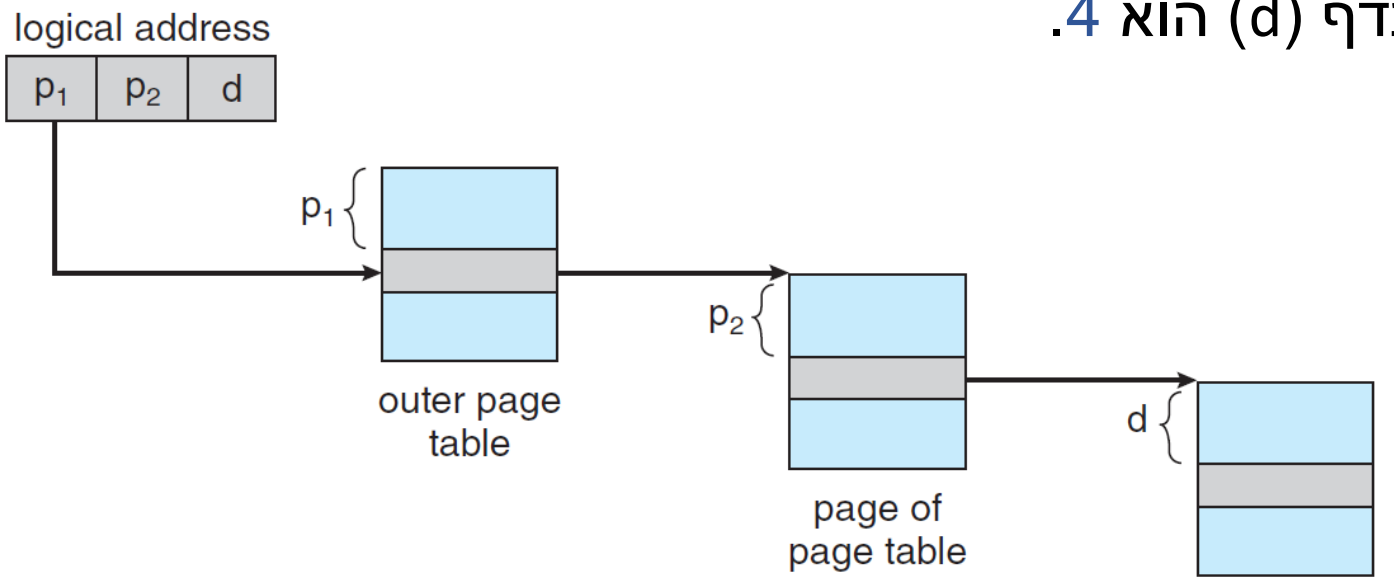
4KB * 4 = 16KB

דוגמה, תרגום כתובת במפת דפים היררכית

- נניח כתובות של 32 ביטים, כדי לבנות את טבלת הדפים בשתי רמות, נחלק את הכתובת לשלושה חלקים.

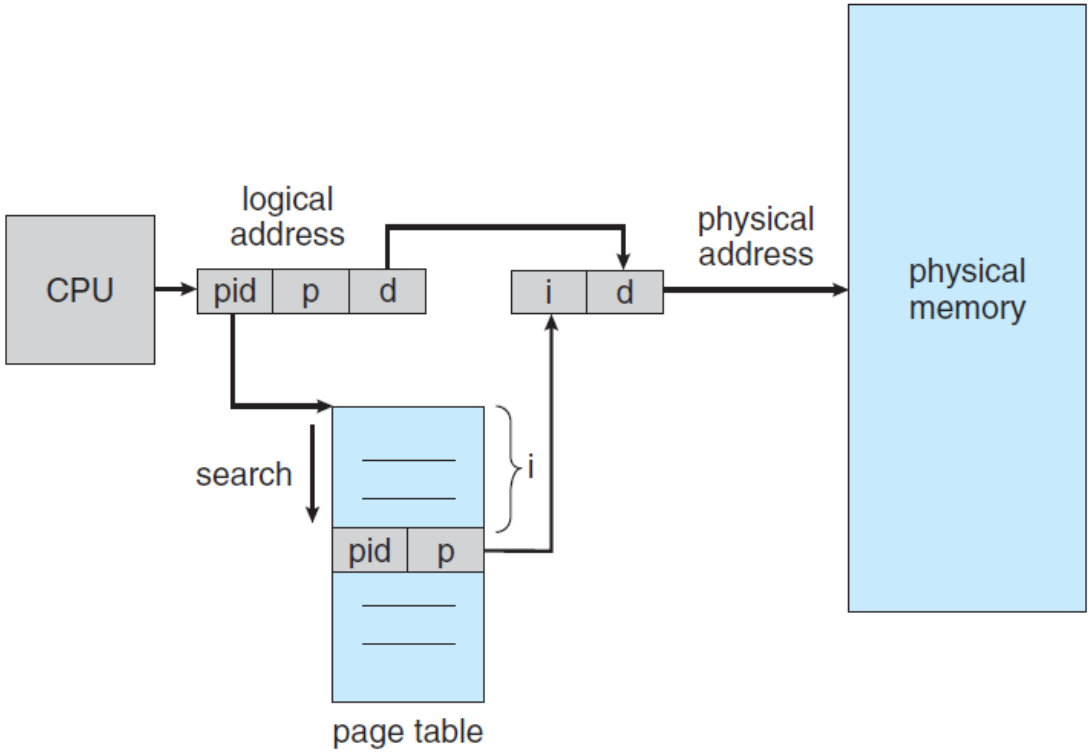


- כדי לתרגם את הכתובת, נשתמש בחלק הראשון (p1) כאינדקס לטבלה ברמה העליונה, בחלק השני (p2) כאינדקס לטבלה ברמה השנייה והחלק השלישי (d) הוא המקום בדף (offset).
- לדוגמה, בכתובת 0x00403004 האינדקס לרמה העליונה (p1) הוא 1, האינדקס לרמה השנייה הוא 3, והמקום בדף (d) הוא 4.



פתרון נוסף: מפת דפים הפוכה

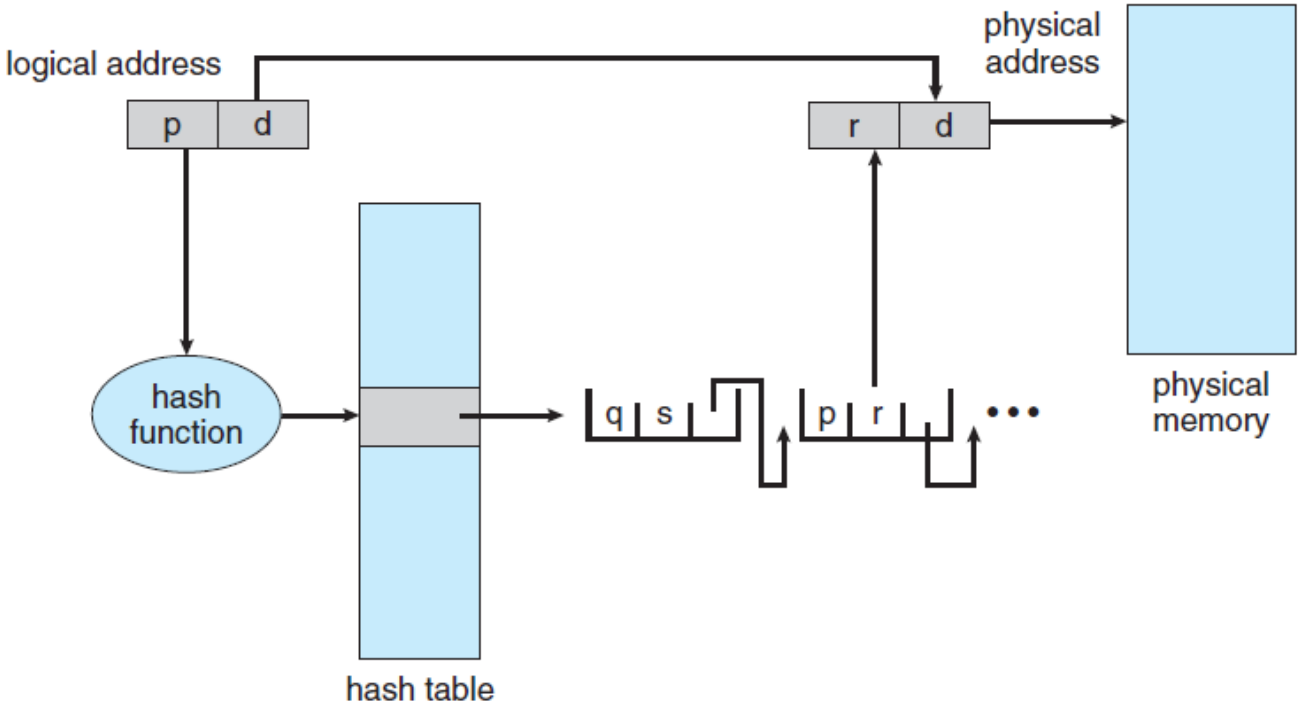
- **בעיה:**
  - בכתובות של 64 ביטים, בטבלה היררכית יהיה צורך ברמות נוספות ובגישות נוספות לזיכרון.
  - טבלת דפים **לכל תהליך** תופסת מקום.
- **פתרון**: במקום טבלת דפים **לכל תהליך** שממפה **כתובות לוגיות לפיסיות**, תהיה טבלה **אחת** במערכת שממפה **כתובות פיסיות ללוגיות**.



- מספר הכניסות בטבלה הוא כמספר המסגרות בזיכרון הפיסי המותקן המחשב.
- דוגמה: זיכרון פיסי 1GB, גודל דף 2KB, מספר הכניסות $2^{30} / 2^{11} = 2^{19} = 512K$
- כל כניסה בטבלה תכיל את מספר התהליך שהדף הפיסי שייך לו, מספר הדף הלוגי של התהליך וביטים לבקרה.
- **בעיה:** מאחר שה- MMU צריך לתרגם כתובת **לוגית לפיסית** יהיה צורך בחיפוש לינארי כדי למצוא את הכתובת הלוגית.

טבלת גיבוב למפת דפים הפוכה

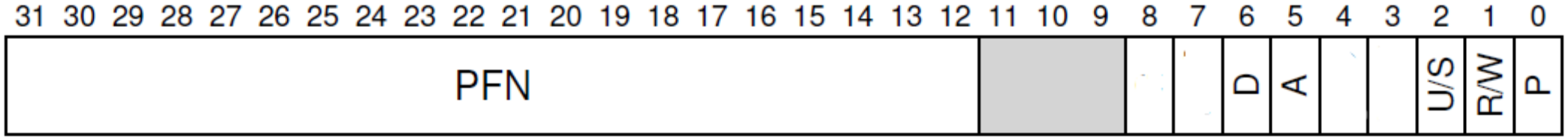
- כדי שהחיפוש יהיה מהיר משתמשים בטבלת גיבוב.
- כל כניסה בטבלה מכילה מצביעה לרשימת הדפים הלוגיים שפונקציית הגיבוב ממפה לאותו מספר דף פיסי.
- כל איבר ברשימה מכיל מספר תהליך, מספר דף לוגי ומספר דף פיסי שניתן לתהליך.
- כאשר המיפוי לא נמצא במטמון, מערכת ההפעלה תחפש בטבלת הגיבוב ותכניס למטמון.



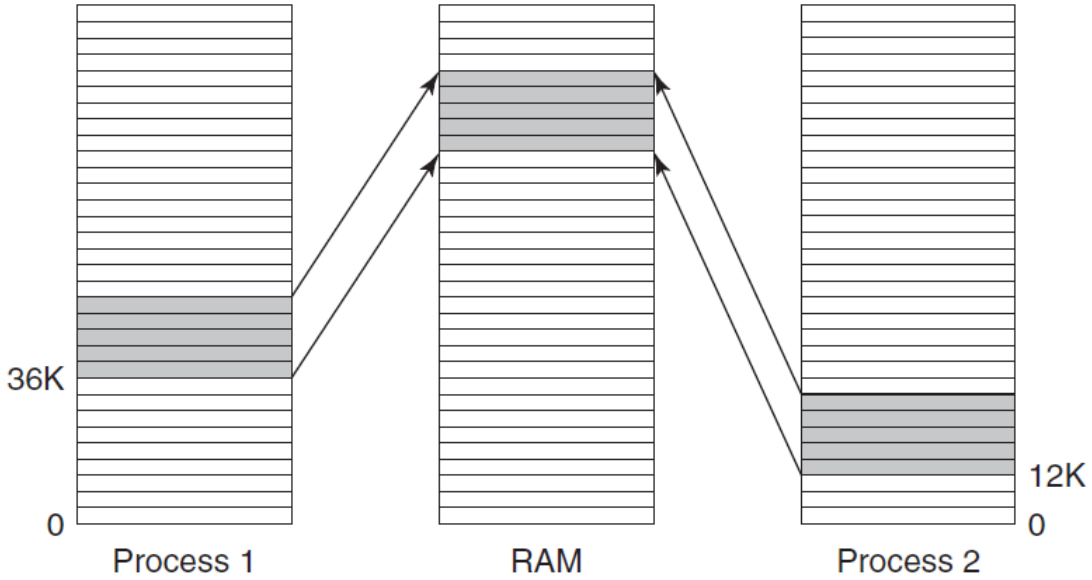
מבנה כל כניסה בטבלת הדפים - PTE

- נניח שוב כתובות בגודל 32 ביט וגודל דף 2^{12}=4K, אם כן יכולים להיות כמיליון (2^{20}) דפים פיסיים במחשב.
- נניח שגודל כניסה 32 ביטים (4 בתים), כדי לציין את הדף הפיסי (PFN) צריך 20 ביטים, נשארו עוד 12 ביטים:
  - Present – האם הדף בזיכרון הפיסי, אחרת החומרה תיצור פסיקה.
  - Read/Write – האם אפשר לכתוב לדף, אחרת החומרה תיצור פסיקה.
  - User/Supervisor – האם משתמש רגיל יכול לגשת לדף.
  - Accessed – האם נגשו לדף לאחרונה, משמש בהחלטה להחלפת דף (להלן).
  - Dirty – האם כתבו לדף, משמש בהחלטה להחלפת דף.
- הביטים של ההרשאות מועתקים למטמון.

An x86 Page Table Entry (PTE)



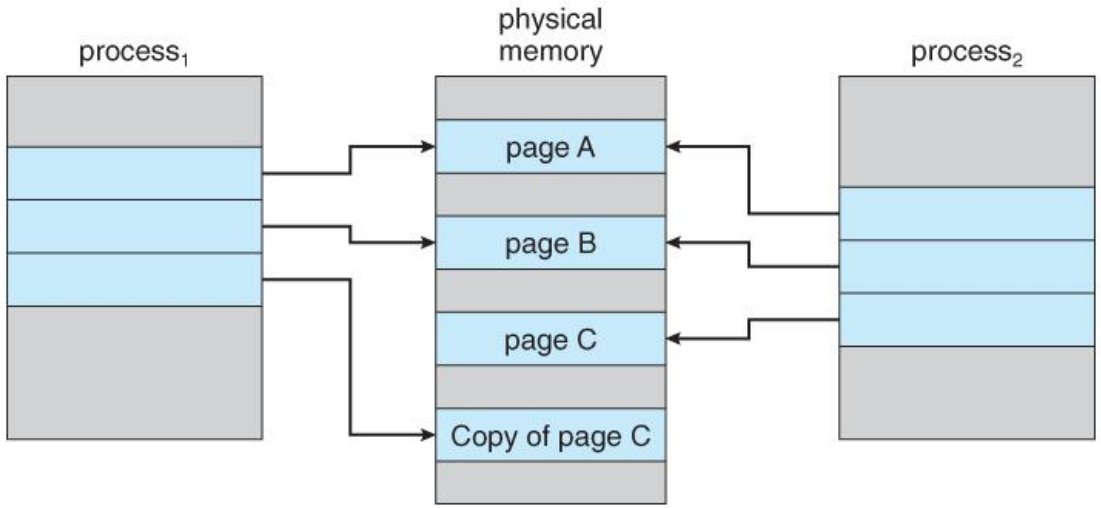
שיתוף דפים



- אחד היתרונות של חלוקת הזיכרון לדפים הוא האפשרות לשתף דפים בין תהליכים.
- שיתוף נתונים: אפשר ליצור שיתוף דפים בין תהליכים לצורך העברת נתונים (IPC).
- שיתוף קוד: לדוגמה, הרבה תהליכים משתמשים בספריה הסטנדרטית של C.
  - אפשר לטעון רק עותק אחד של דפי **הקוד** של הספרייה לזיכרון הפיסי.
  - וליצור מיפוי בכל טבלאות הדפים של המשתמשים בספריה לדפים פיסיים אלו.
  - לכל תהליך יהיה עותק נפרד של דפי **המשתנים** של הספרייה.
  - מאחר שקוד הספרייה ממופה בכתובות לוגיות שונות, הקוד לא מכיל כתובות מוחלטות אלא כתובות יחסיות (PIC).

דחית העתקה באמצעות שיתוף דפים - copy-on-write (COW)

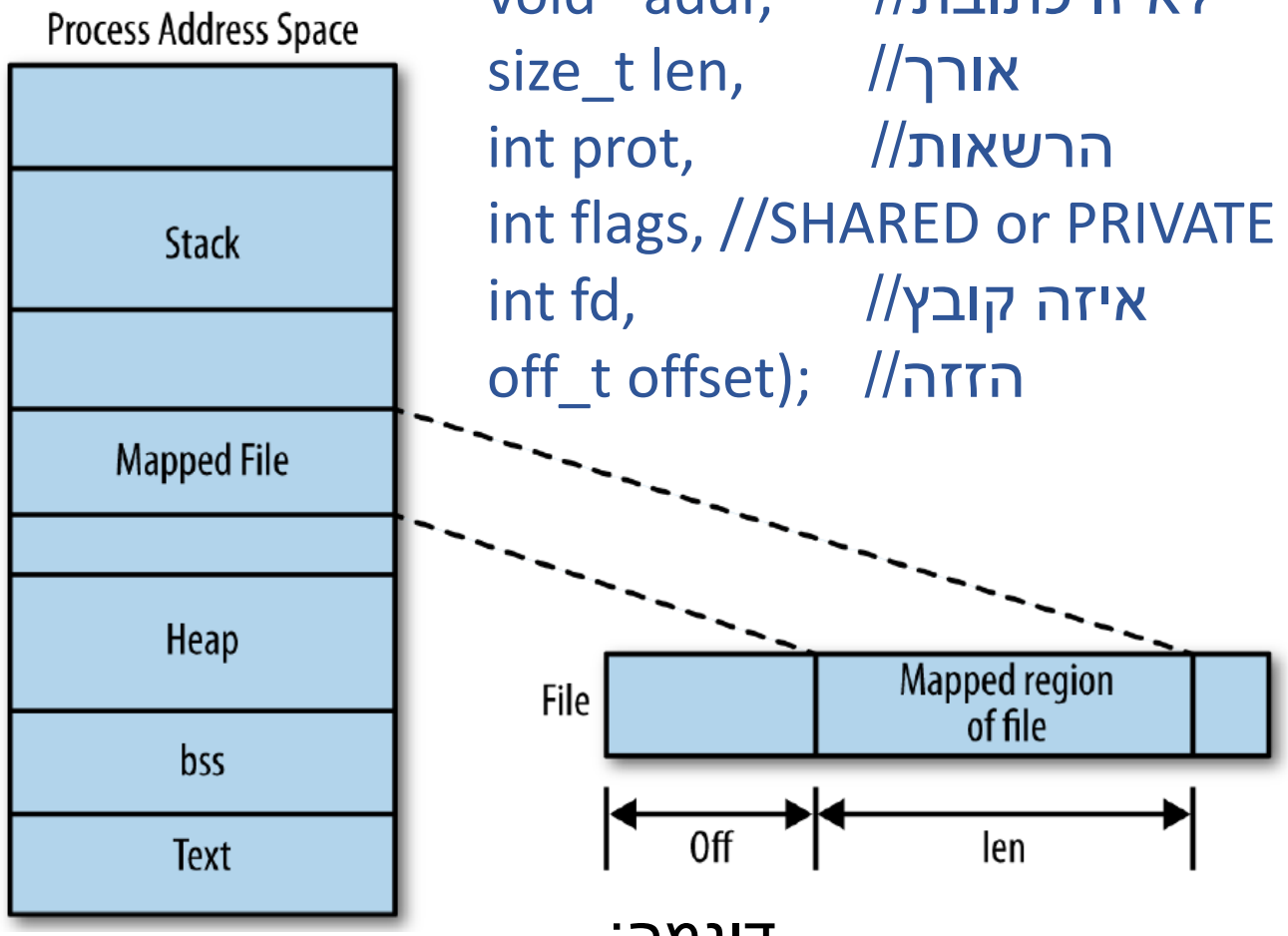
- fork יוצר תהליך ילד שבו קטעי הזיכרון הם העתק של קטעי הזיכרון של ההורה.
  - בדרך כלל הילד מבצע exec מיד אחרי ה- fork ומחליף את קטעי הזיכרון, אם כן ביצוע ההעתקה היה מיותר.
- במקום להעתיק את קטעי הזיכרון אפשר לשתף אותם בין ההורה לילד, לסמן אותם read-only בטבלאות הדפים של ההורה והילד ולזכור שהם copy-on-write.
  - אם הילד ביצע exec, חסכנו את ההעתקה.
  - אם אחד מהם ינסה לכתוב לאחד הדפים המשותפים, זה יגרום לפסיקה.
  - מערכת ההפעלה תשים לב שהשיתוף הוא מסוג copy-on-write ובמקום להפסיק את התהליך שגרם לפסיקה תעתיק את הדף.



מיפוי קבצים לדפי זיכרון – memory mapped files

- כשקוראים או כותבים קבצים עם read() או write(), בכל קריאה או כתיבה:
  - מתבצע מעבר למערכת ההפעלה וחזור.
  - יש צורך להעתיק מהקרנל למשתמש או להפך.
- קריאה ל- mmap() ממפה דפי זיכרון לוגי לבלוקים של קובץ.
- הקריאה והכתיבה לקובץ מתבצעות על ידי קריאה וכתיבה של מערך בתים בזיכרון (כפי שקוראים וכותבים משתנים).
- גישה לדף תגרום לפסיקת דף שתביא את הבלוק לזיכרון הפיסי.

```
void * mmap (          //מחזיר כתובת
    void *addr,        //לאיזו כתובת
    size_t len,        //אורך
    int prot,          //הרשאות
    int flags, //SHARED or PRIVATE
    int fd,            //איזה קובץ
    off_t offset);     //הזזה
```



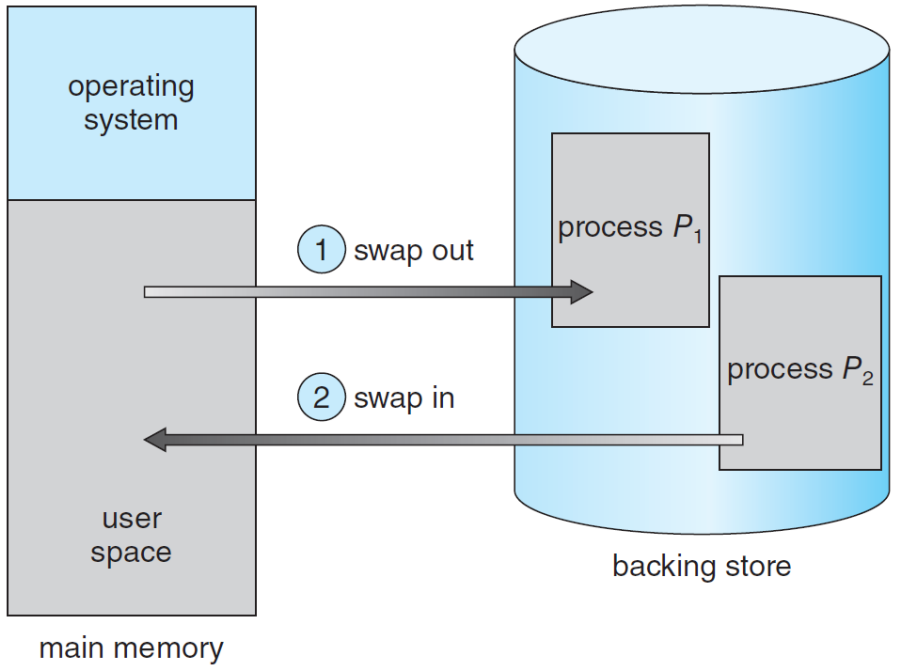
דוגמה:

```
p = mmap (0, len, PROT_READ, MAP_SHARED, fd, 0);
```

mmap()

```
int main () {
        int size = 67; // size of mapped file
        int i;
        char *p;
        int fd;
        fd = open ("fruits.txt", O_RDONLY);
        p = mmap (0, size, PROT_READ, MAP_SHARED, fd, 0);
        for (i = 0; i < size; i++)
                putchar (p[i]);
        munmap (p, size);
        return 0;
}
```

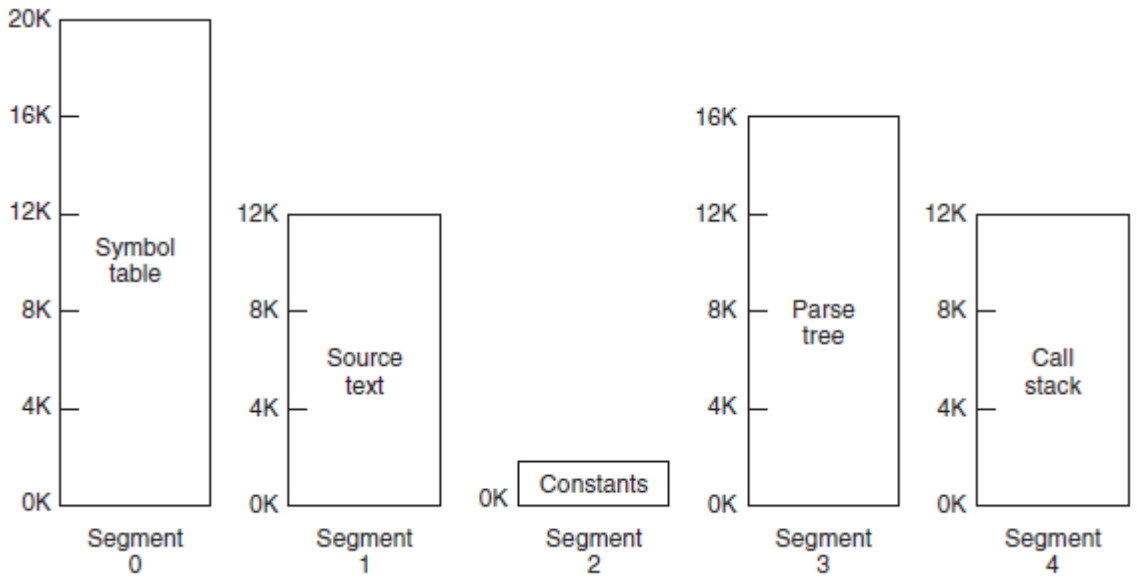
Swapping

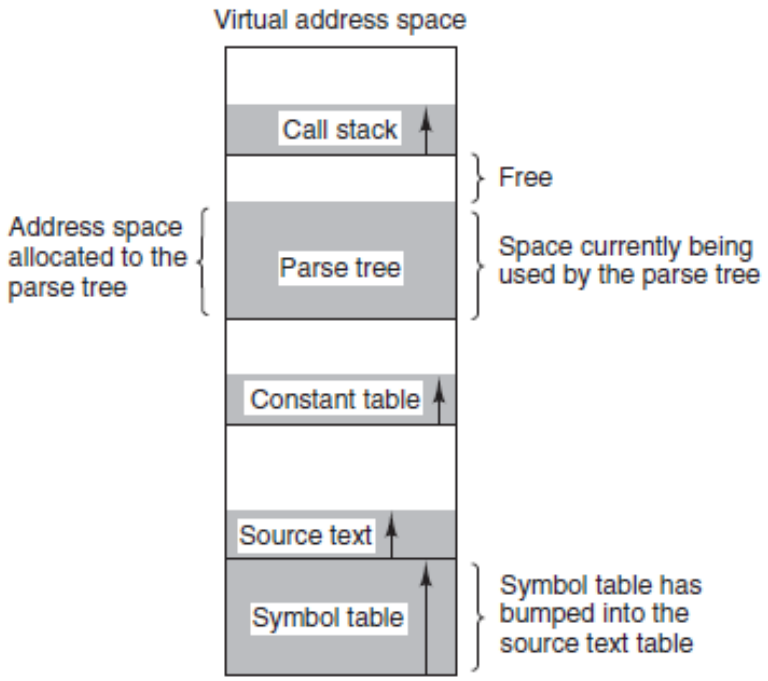


- זיכרון יכול להכיל מספר מסוים של תהליכים.
- כדי להריץ מספר יותר גדול, אפשר להעביר את קטעי הזיכרון של תהליך לדיסק (swapping) ולהחזיר להמשך ריצה.
  - המערכת תבחר להעביר לדיסק תהליכים שהם במצב המתנה (sleep).
  - אפשר להעביר את כל דפי התהליך או רק חלק מהדפים.
- המקום בדיסק (swap space)
  - המקום בדיסק יכול להיות קובץ במערכת הקבצים.
  - זה איטי כי צריך לעבור בהיררכיה של התיקיות ולקרוא את הבלוקים המפוזרים בדיסק.
  - כדי שיהיה יותר מהיר משתמשים במחיצה נפרדת.
  - במחיצה זו אין תיקיות והקצאת הבלוקים רציפה.

חלוקה לקטעים - Segmentation

- במקום לחלק את הזיכרון לדפים בגודל שווה, אפשר לחלק לקטעים בגודל שונה, קטעים אלו יתאימו לחלקים לוגיים של התכנית.
- לדוגמה, קומפיילר יוצר קטעי זיכרון שונים כדי לתרגם תכנית:
  - טבלת סמלים, קוד המקור, קבועים, עץ פיסוק, מחסנית.
- במקום להקצות את החלקים באופן רציף בזיכרון, אפשר להקצות את החלקים בקטעים שונים שכל אחד מהם מתחיל בכתובת לוגית 0.

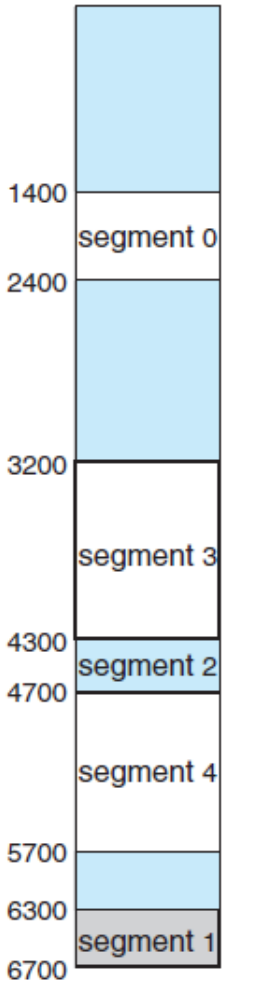




חלוקה למקטעים - Segmentation

	limit	base
0	1000	1400
1	400	6300
2	400	4300
3	1100	3200
4	1000	4700

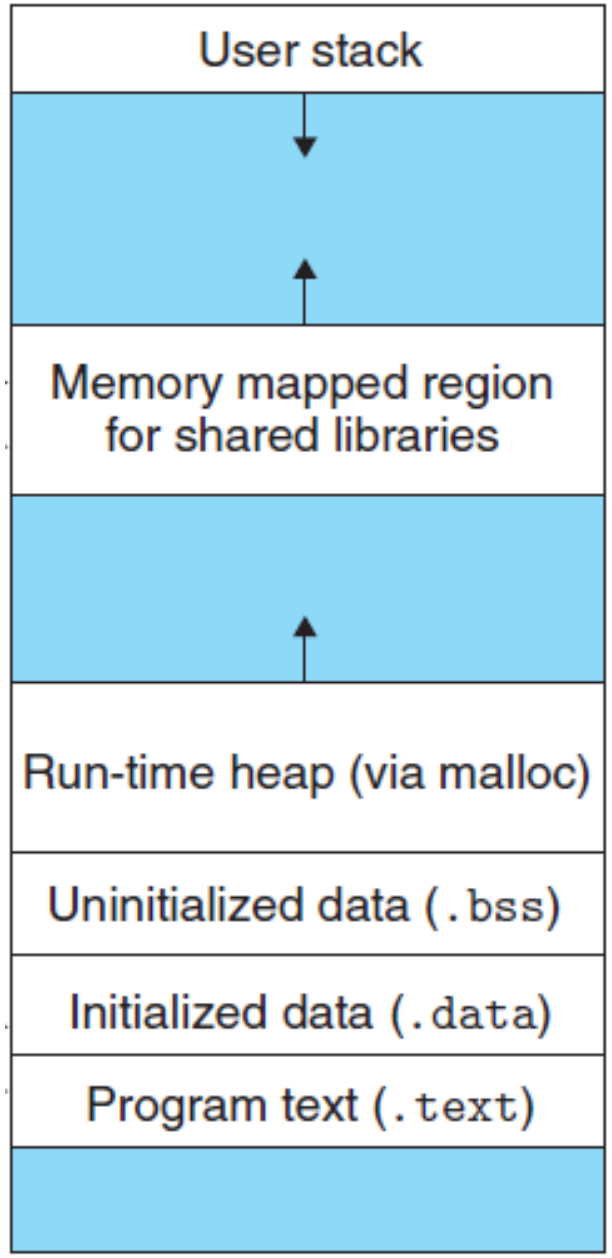
segment table



physical memory

- יתרונות:
  - מאפשר להגדיל את מרחב הזיכרון של תכנית מעבר לגודל האוגר.
  - כל קטע יכול לגדול ללא תלות בקטעים אחרים.
  - מפשט קומפילציה וקישור בין חלקי תכנית.
  - מאפשר שיתוף והגנה על חלקים לוגיים של התכנית.
- חסרונות:
  - דורש התערבות התכניתן.
- בשיטת הקטעים כל כתובת מורכבת מזוג שמציין את הקטע וההזזה בקטע.
- מיפוי כתובות לוגיות לכתובות פיסיות יעשה על ידי החומרה (MMU) באמצעות טבלת מקטעים שתכיל אוגר בסיס ואוגר גבול **לכל קטע**.
- מעבד X86 משתמש בקטעים כאשר כל קטע מחולק לדפים.
- במעבד X86-64 אין שימוש בקטעים (רק במצב תאימות לאחור).

זיכרון וירטואלי

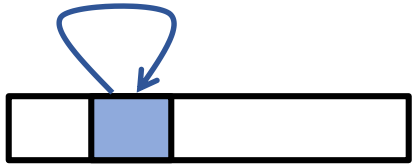


- עד כה הנחנו שיש צורך לטעון את כל התכנית לזיכרון כדי להריץ אותה.
  - אם התכנית גדולה מהזיכרון הפיסי, אי אפשר להריץ אותה.
  - אם רוצים להריץ כמה תכניות צריך מקום לכולם או להשתמש ב- swapping.
- בזיכרון וירטואלי, **מרחב הזיכרון** של התהליך הרץ הוא כפי מרחב הכתובות שאפשר להגיע אליהם באמצעות ה- CPU.
  - דוגמה: במחשב 32 ביטים אפשר להגיע ל- $4G=2^{32}$ כתובות, אם כן כל תהליך מקבל 4G זיכרון וירטואלי.
  - זיכרון וירטואלי מאפשר לתת לכל תהליך אותו מבנה של זיכרון (קוד, נתונים, מחסנית), זה מפשט קומפילציה וטעינה לזיכרון.
  - זיכרון וירטואלי מבוסס על כך שתהליך צריך רק חלק מדפי התכנית כדי לרוץ, את החלק הזה מביאים בזמן ריצה לזיכרון הפיסי (demand paging) ומעדכנים את טבלת הדפים.
  - כשהתהליך עובר לבצע במקום אחר, אפשר לפנות את הדפים הקודמים ולהביא במקומם דפים אחרים.

תכונת המקומיות של תכניות

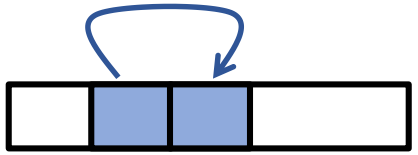
- אם תהליך היה פונה לדפים באופן אקראי, זיכרון וירטואלי היה איטי כי לעיתים קרובות היה צורך לפנות דפים ולהביא דפים אחרים במקומם.
- אבל תהליך פועל לפי תכונת המקומיות, כלומר נשאר לזמן מה בדפים בהם היה לאחרונה.
  - **תכונת המקומיות בזמן:**

אם תהליך פנה לכתובת מסוימת כדי לקרוא פקודה או כדי לקרוא או לכתוב נתון, סיכוי טוב שהוא יפנה שוב לאותה פקודה או נתון **בזמן הקרוב**.



- **תכונת המקומיות במקום:**

אם תהליך פנה לכתובת מסוימת כדי לקרוא פקודה או כדי לקרוא או לכתוב נתון, סיכוי טוב שהוא יפנה **לכתובות סמוכות**.



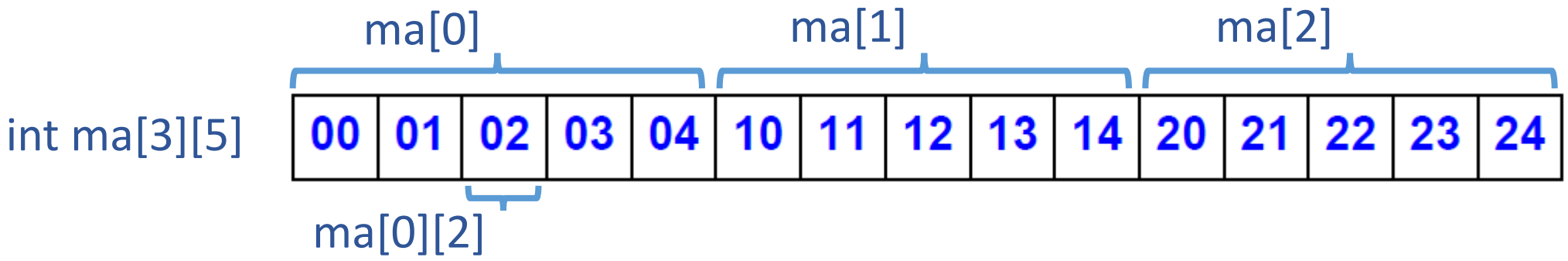
דוגמה למקומיות

```
sum = 0;
for (i = 0; i < n; i++)
   sum += a[i];
return sum;
```

- מקומיות בפקודות:
  - חזרה על אותן פקודות בלולאה. (מקומיות בזמן)
  - ביצוע פקודות לפי סדר. (מקומיות במקום)
- מקומיות בנתונים:
  - כתיבה למשתנה sum בכל חזרה בלולאה. (מקומיות בזמן)
  - קריאה אברי המערך a לפי סדר. (מקומיות במקום)

מקומיות

- שאלה: להלן שתי פונקציות למציאת סכום האברים במטריצה, איזו מהן יותר מקומית?
  - קוד יותר מקומי עשוי לגרום לפחות פסיקות דף ולרוץ יותר מהר.
- הערה: אברי מטריצה מסודרים בזיכרון שורה אחר שורה:



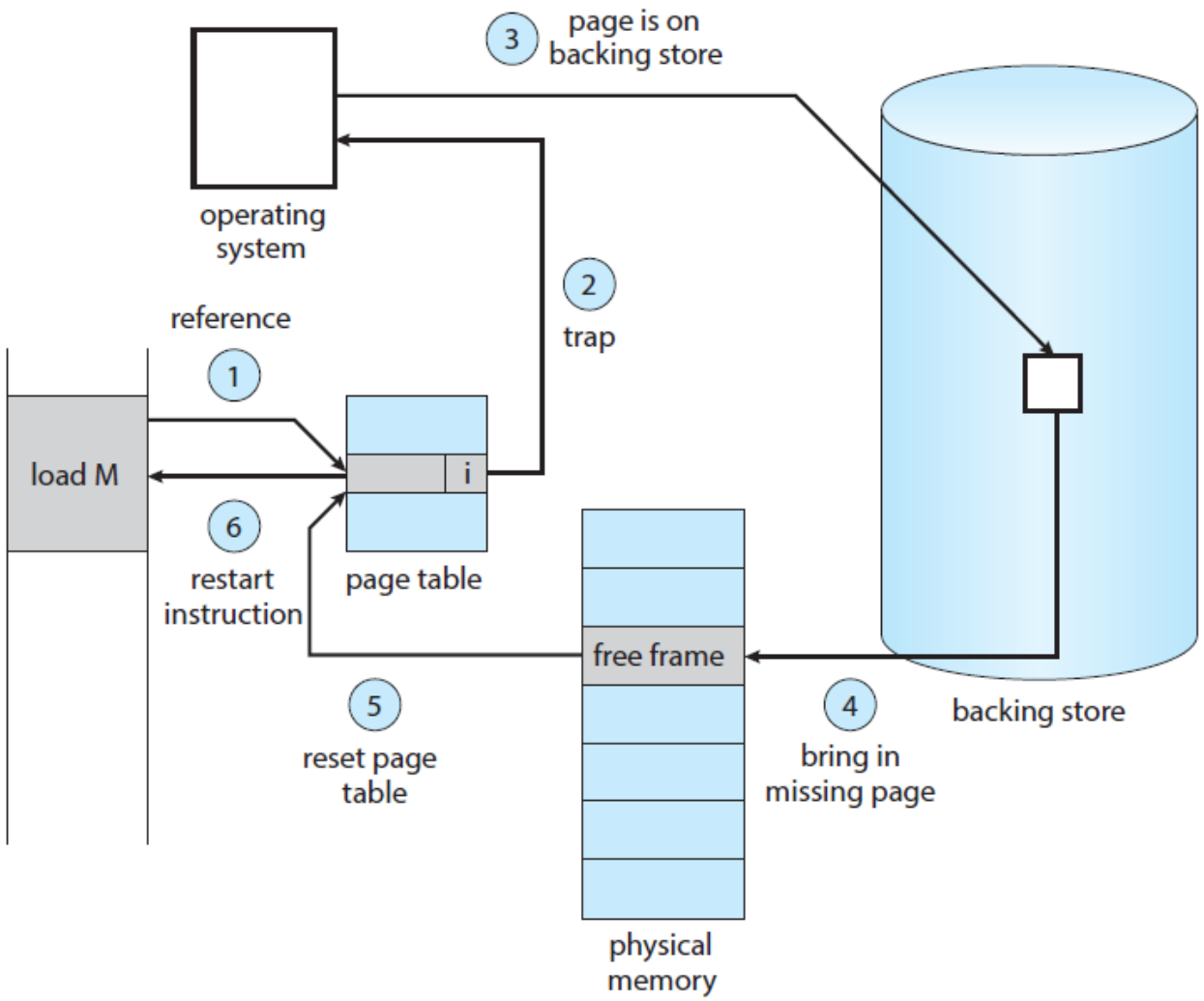

```
int sum_array(int a[M][N])
{
    int i, j, sum = 0;

    for (j = 0; j < N; j++)
        for (i = 0; i < M; i++)
            sum += a[i][j];
    return sum;
}
```

```
int sum_array(int a[M][N])
{
    int i, j, sum = 0;

    for (i = 0; i < M; i++)
        for (j = 0; j < N; j++)
            sum += a[i][j];
    return sum;
}
```

פסיקת דף – page fault

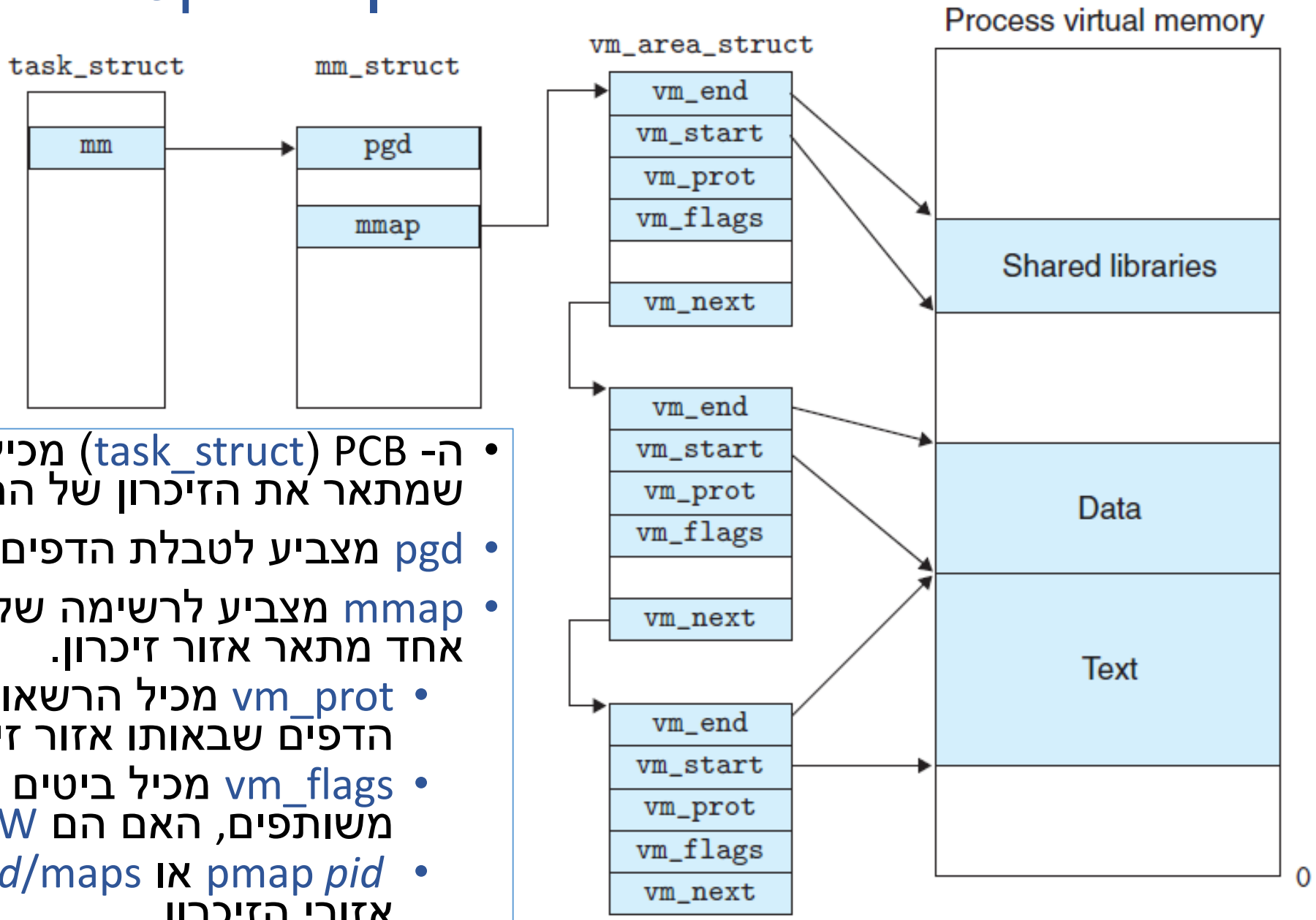


- בטבלת הדפים ישנו ביט present שאומר האם הדף נמצא בזיכרון הפיסי.
- מאחר שבזיכרון וירטואלי לא כל הדפים של תהליך נטענים לזיכרון הפיסי, אם כן לדף שאינו נמצא יכולות להיות שתי משמעויות: הוא **לא במרחב הלוגי** של התכנית או שרק עדיין **לא נטען לזיכרון**.
- בשני המקרים, ניסיון לגשת לאותו דף יגרום **לפסיקה** (page fault).
- בטיפול בפסיקה, מערכת ההפעלה תבדוק ב- PCB של התהליך את רשימת אזורי הזיכרון ולפי זה תדע האם להפסיק את התהליך או להביא את הדף החסר.
- לאחר הבאת הדף יש צורך להפעיל את הפקודה שגרמה לפסיקה מחדש.

אזורי זיכרון בלינוקס

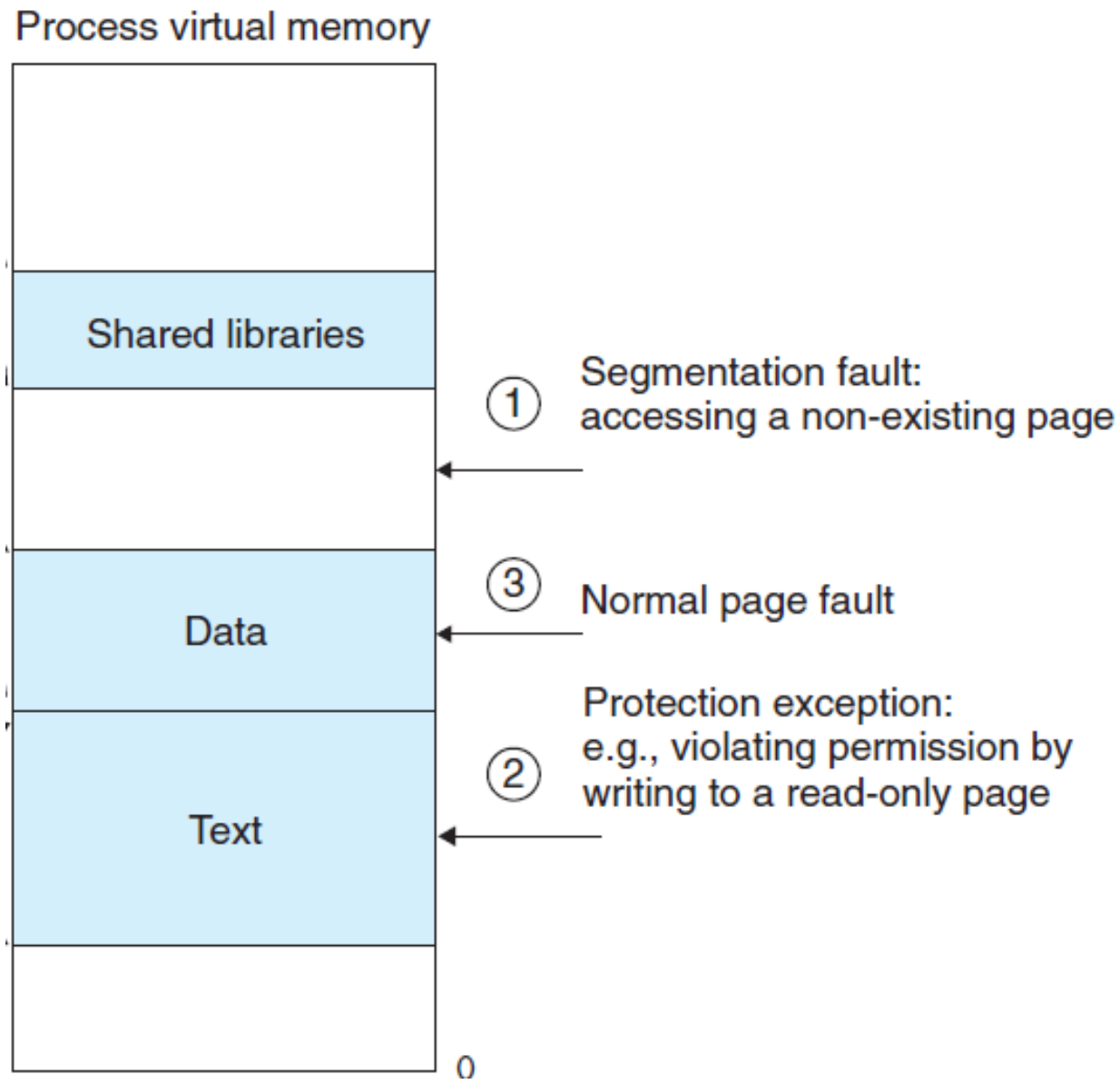
- לינוקס מחלק את זיכרון התהליך לאזורי זיכרון:

code, data, heap, shared library, user stack



- ה- PCB (task_struct) מכיל מצביע ל- mm_struct שמתאר את הזיכרון של התהליך.
- pgd מצביע לטבלת הדפים.
- mmap מצביע לרשימה של vm_area_struct, כל אחד מתאר אזור זיכרון.
  - vm_prot מכיל הרשאות (read/write) של הדפים שבאותו אזור זיכרון.
  - vm_flags מכיל ביטים שאומרים האם הדפים משותפים, האם הם COW, ועוד.
  - *pid* pmap או cat /proc/*pid*/maps מציגים אזורי הזיכרון

טיפול בפסיקת דף לפי אזורי הזיכרון



- הקוד המטפל בפסיקת דף מבצע את הצעדים הבאים:
  1. האם הכתובת אינה כלולה באחד מאזורי הזיכרון של התהליך, אם כן הכתובת אינה במרחב הלוגי של התכנית, קוד הפסיקה ישלח לתהליך segmentation fault
  2. הכתובת כלולה באחד מאזורי הזיכרון, אך האם הייתה הרשאת גישה לדף (read, write, or execute)
     אם לא, קוד הפסיקה ישלח לתהליך segmentation fault
  3. כעת ברור שהגישה הייתה חוקית, מביאים את הדף החסר ומעדכנים את טבלת הדפים.

החלפת דף – page replacement

- כאשר מתרחשת פסיקת דף, יש צורך לטעון את הדף החסר לזיכרון הפיסי.
- נניח שהוקצו עבור תהליך מספר מסוים של דפים פיסיים והם תפוסים, אזי יש צורך להחליף תוכן של אחד הדפים בתוכן של הדף החסר.
- באיזה דף נבחר לצורך ההחלפה?
  - רצוי לבחור בדף שאין בו צורך בקרוב, כדי שבהמשך מספר הפסיקות יהיה הנמוך ביותר.
- נבדוק כמה אפשרויות: OPT , FIFO , LRU.
- כדי להשוות את האפשרויות, נבדוק את מספר הפסיקות בגישה לסדרה מסוימת של דפים.
  - דוגמה, נניח גודל דף של 100 בתים וגישה לסדרת הכתובות:

0100, 0432, 0101, 0612, 0102, 0103, 0104, 0101, 0611, 0102, 0103

- נחלק ב- 100 כדי לקבל את מספר הדף, כמו כן אם כתובות עוקבות ניגשות לאותו דף לא תגרם פסיקה ואפשר לצמצם לגישה אחת.
  - נקבל את סדרת מספרי הדפים:

1, 4, 1, 6, 1, 6, 1, 6, 1

החלפת דף OPT

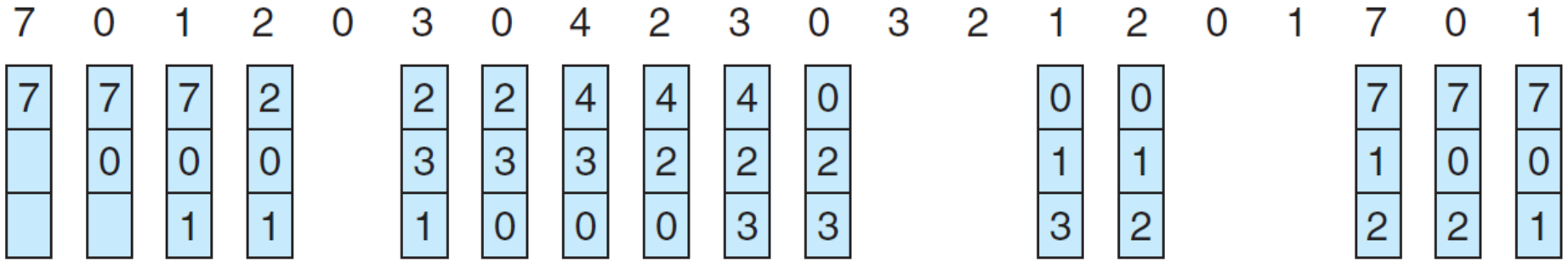
- החלפת דף אופטימלית (Optimal) היא החלפה שיש לה את מספר הפסיקות הנמוך ביותר האפשרי.
- האלגוריתם הוא: החלף את הדף שלא **יהיה** בשימוש בזמן הארוך ביותר.
- בדוגמה למטה יש 3 דפים פיסיים, בהחלפה אופטימלית יהיו 6 פסיקות בנוסף ל- 3 פסיקות הכרחיות.
- החלפה אופטימלית אינה מעשית כי היא מצריכה ידיעת העתיד, אבל יכולה לשמש לצורך השוואה לדרכים אחרות.

7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
7	7	7	2		2		2			2			2				7		
	0	0	0		0		4			0			0				0		
		1	1		3		3			3			1				1		

החלפת דף FIFO

- האלגוריתם הוא: החלף את הדף **שהובא** מוקדם ביותר (first-in, first-out) לזיכרון הפיסי.
- כדי לדעת את סדר הבאת הדפים אפשר ליצור תור (queue) עם מספרי הדפים הפיסיים, כל דף חדש יכנס לסוף התור ואת הדף להחלפה ניקח מראש התור.

- אם נחזור על הדוגמה הקודמת, בהחלפת FIFO יהיו 12 פסיקות (פי שניים מהחלפת OPT) בנוסף ל- 3 פסיקות הכרחיות.
- ההיגיון ב- FIFO הוא שאם הבאנו את הדף בעבר הרחוק, כנראה שכעת אין בו צורך.
  - מצד שני, יתכן שזה משתנה שמשתמשים בו לכל אורך התכנית.



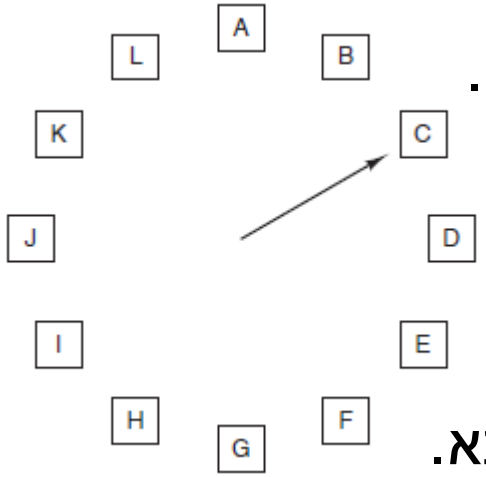
החלפת דף LRU

- האלגוריתם הוא: החלף את הדף **שלא היה בשימוש** בזמן הרב ביותר (least recently used).
- LRU הוא קירוב של OPT מתוך הנחה שאם לא השתמשנו בדף בעבר הקרוב, לא נשתמש בו בעתיד הקרוב.
- אם נחזור על הדוגמה הקודמת, בהחלפת LRU יהיו 9 פסיקות (פי אחד וחצי מהחלפת OPT) בנוסף ל- 3 פסיקות הכרחיות.
- כדי לממש LRU צריך לזכור מתי השתמשו לאחרונה בכל דף, זה צריך להיעשות בכל פניה לזיכרון!
- לא מעשי לבצע פסיקה שתעדכן זמן בכל פניה לזיכרון.

7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
7	7	7	2		2		4	4	4	0			1		1		1		
	0	0	0		0		0	0	3	3			3		0		0		
		1	1		3		3	2	2	2			2		2		7		

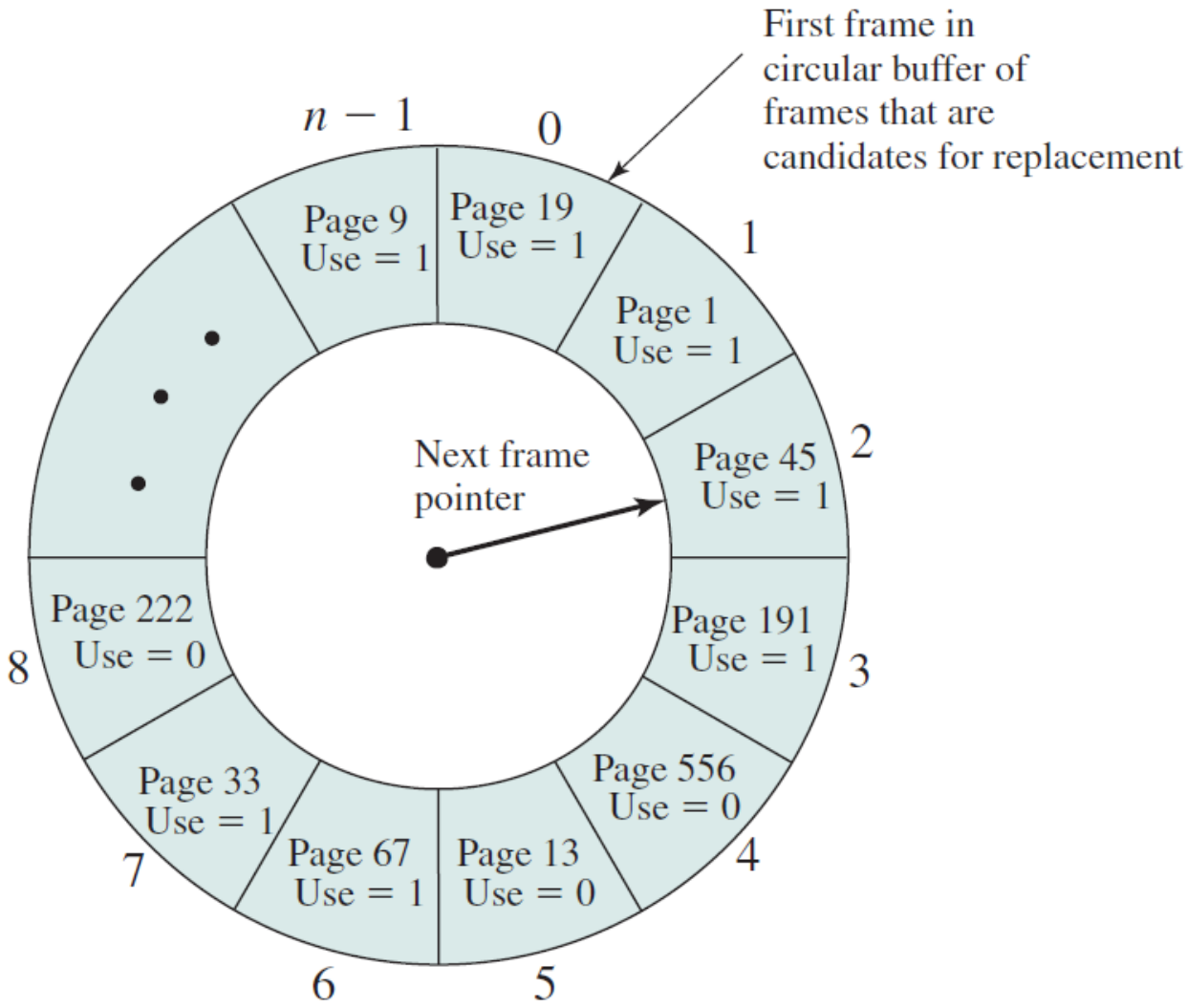
אלגוריתם השעון

- מאחר שלא מעשי לממש LRU אמיתי, משתמשים בקירוב ל- LRU שנקרא אלגוריתם השעון.
- אלגוריתם השעון משתמש בביט Accessed שבטבלת הדפים (שאומר האם נגשו לדף).
  - הביט לא מאפשר לדעת את סדר הגישה לדפים אלא רק אם נגשו אליהם לאחרונה.
  - כשדף נטען לזיכרון ערך הביט Accessed הוא 1 - כעת נגשו אליו.
  - ישנו תור מעגלי (סוף התור מקושר לראש התור) של מספרי הדפים הפיסיים.
  - ישנו מצביע למקום הנוכחי בתור המעגלי - מחוג השעון.

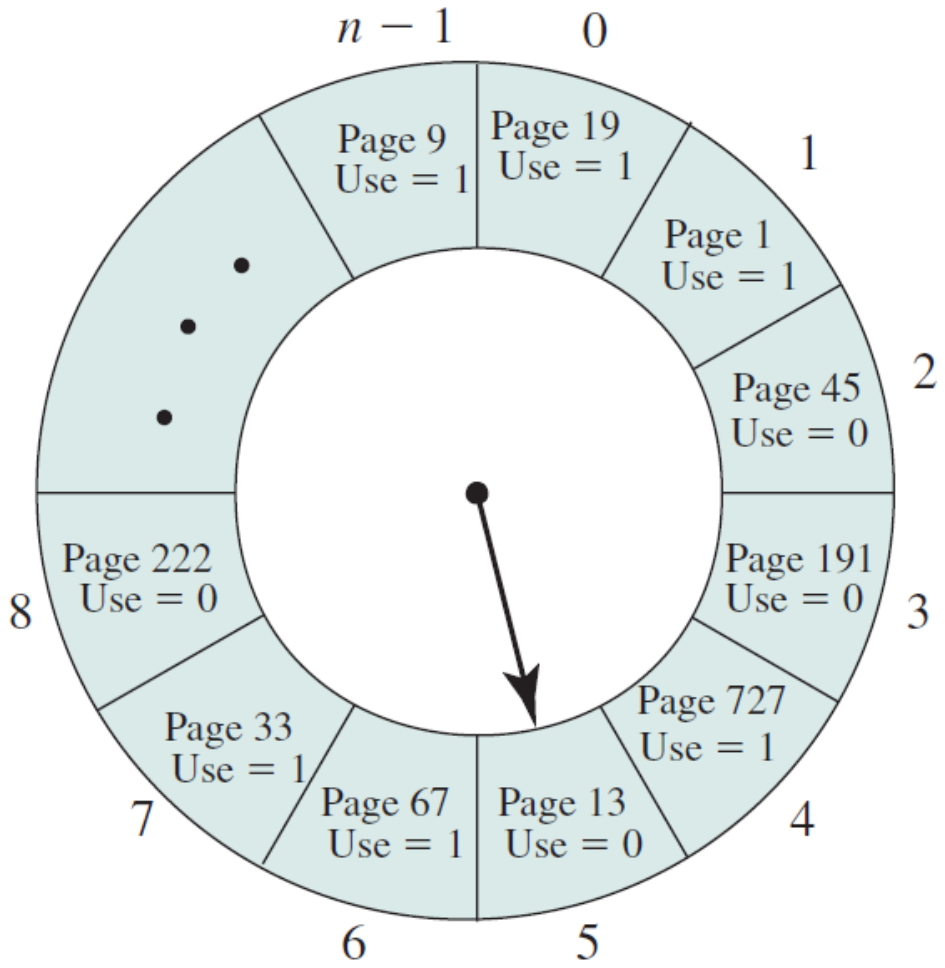


- כשצריך לבחור דף להחלפה:
  - בודקים את הביט Accessed של הדף שהמחוג מצביע עליו.
  - אם ערכו 0 הוא משמש להחלפה, ואז ערכו הופך ל- 1 והמחוג עובר לדף הבא.
  - אם ערכו 1, הוא לא משמש להחלפה, מערכת ההפעלה משנה את ערכו ל-0 והמחוג עובר לדף הבא.
  - אם עד הסיבוב הבא הדף לא יהיה בשימוש, הביט יישאר 0 והוא ישמש להחלפה.
- אם בכל הדפים הביט הוא 1, יתבצע סיבוב שיהפוך אותם ל- 0, ונקבל החלפת FIFO.

אלגוריתם השעון



(a) State of buffer just prior to a page replacement



(b) State of buffer just after the next page replacement

סיכום אלגוריתמים להחלפת דף

Page address stream	2	3	2	1	5	2	4	5	3	2	5	2
OPT	2	2	2	2	2	2	4	4	4	2	2	2
		3	3	3	3	3	3	3	3	3	3	3
				1	5	5	5	5	5	5	5	5
					F		F			F		
LRU	2	2	2	2	2	2	2	2	3	3	3	3
		3	3	3	5	5	5	5	5	5	5	5
				1	1	1	4	4	4	2	2	2
					F		F		F	F		
FIFO	2	2	2	2	5	5	5	5	3	3	3	3
		3	3	3	3	2	2	2	2	2	5	5
				1	1	1	4	4	4	4	4	2
					F	F	F		F		F	F
CLOCK	2*	2*	2*	2*	5*	5*	5*	5*	3*	3*	3*	3*
		3*	3*	3*	3	2*	2*	2*	2	2*	2	2*
				1*	1	1	4*	4*	4	4	5*	5*
					F	F	F		F		F	