תרגול 8

למה צריך זיכרון וירטואלי?

Paging במעבדי אינטל 32-ביט

Paging במעבדי אינטל 64-ביט

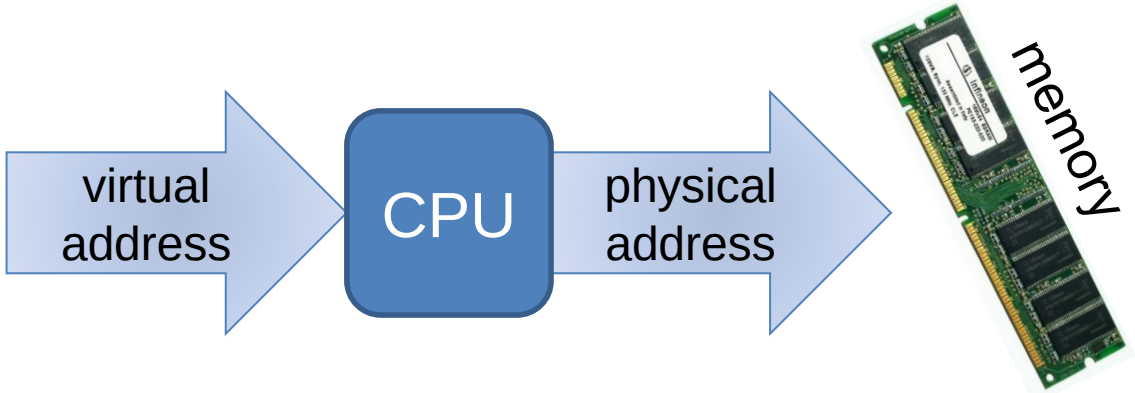
TL;DR

- גישה ישירה לזיכרון הפיזי הייתה יוצרת הרבה בעיות: מחסור בזיכרון רציף, היעדר בידוד בין תהליכים, מגבלה על מרחב הזיכרון האפשרי.

- האבסטרקציה שפותרת את כל הבעיות הללו היא **זיכרון וירטואלי.**

process A:

…
mov **$0x700**, %rax
add %rax, %rbx
…



- אבל אין ארוחות חינם: **זיכרון וירטואלי פוגע בביצועים.**
  - כל פקודת גישה לזיכרון דורשת תרגום יקר: וירטואלי ← פיזי.

למה צריך זיכרון וירטואלי?

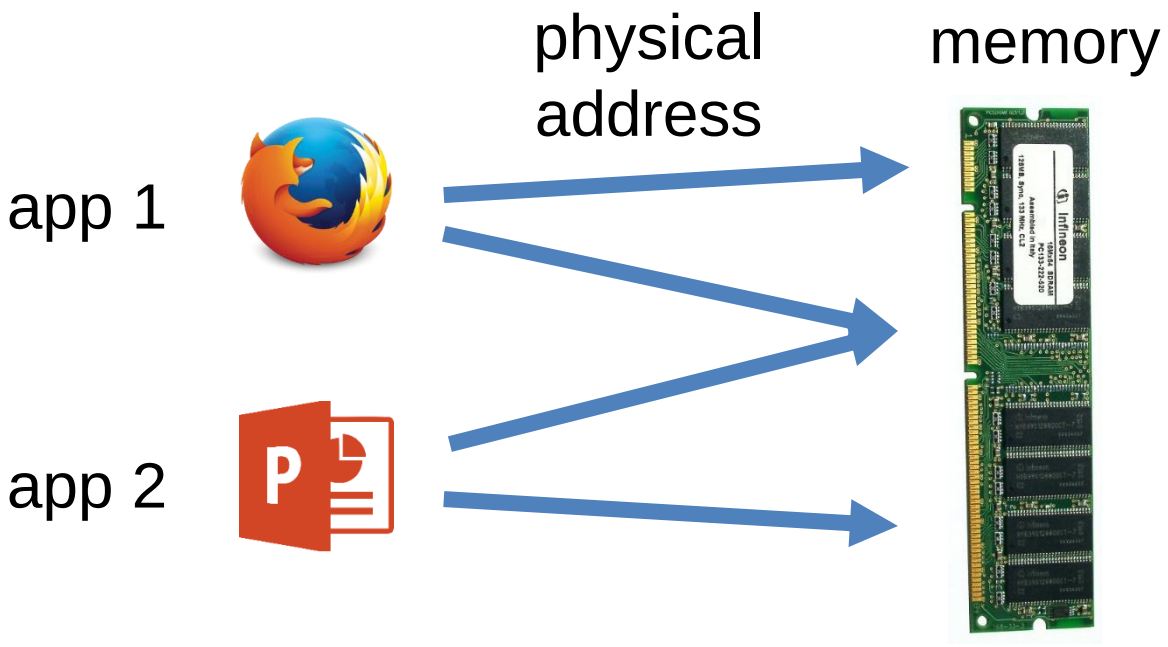
או: מדוע לא ניגשים ישירות לזיכרון הפיזי?

זיכרון פיזי

- התקני האיחסון במחשב נחלקים לשניים:
  - זיכרון (DRAM) – איחסון נדיף, קטן יותר ומהיר יותר (סדר גודל 100ns).
  - דיסק קשיח – איחסון עמיד, גדול יותר ואיטי יותר (סדר גודל 1ms).

- פקודות מכונה יכולות לפעול רק על רגיסטרים ו/או נתונים **בזיכרון**.
  - הקוד המבוצע ע"י המעבד והנתונים הדרושים לביצוע הקוד חייבים להיות בזיכרון בזמן הביצוע.
  - אם רוצים לעבד מידע מהדיסק, יש להביא אותו לזיכרון, לעבד אותו, ולכתוב את התוצאה חזרה לדיסק.
- הגישה לזיכרון היא, בסופו של דבר, באמצעות **כתובות פיזיות** של בתים (bytes). למשל, עבור זיכרון בגודל 8GB הכתובות הפיזיות הן מספרים שלמים בתחום [0, 8G – 1].
- עם זאת, יש חסרונות רבים לגישה ישירה באמצעות כתובות פיזיות.

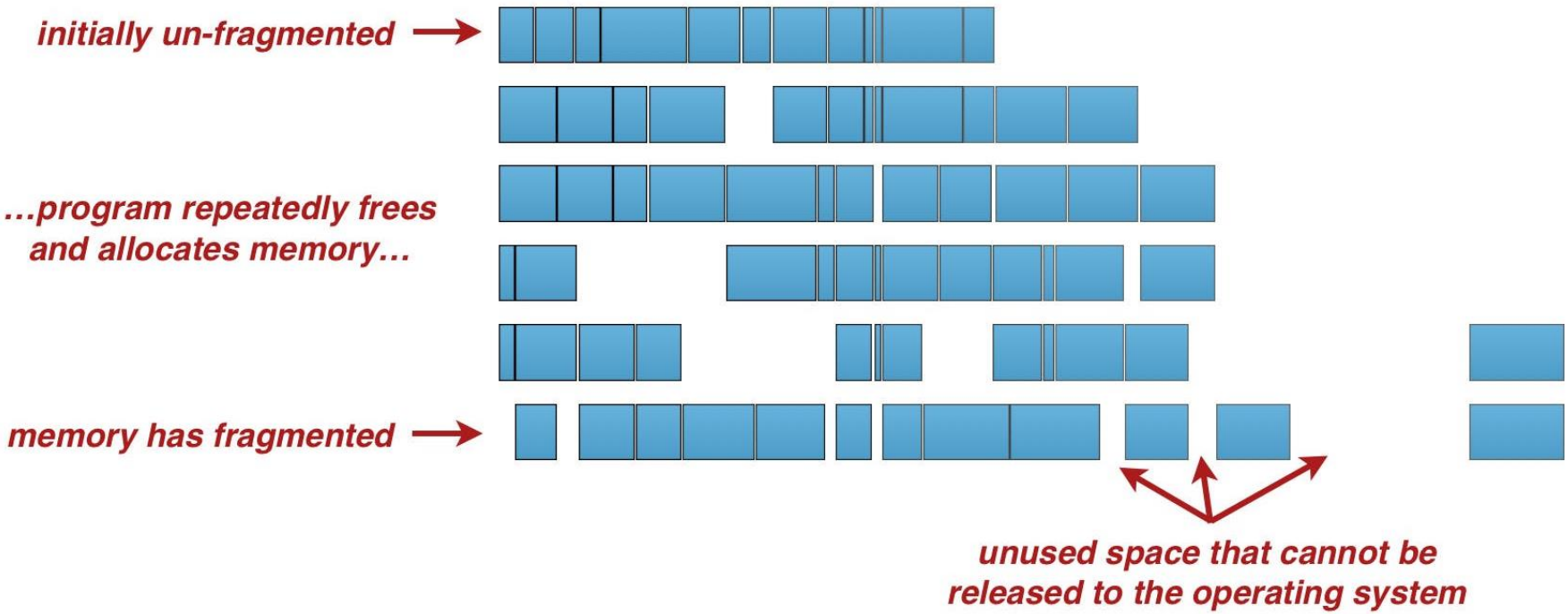
חסרון #1: היעדר בידוד/הגנה בין תהליכים



- אין הגנה על המידע – תהליך א' יכול לגשת לזיכרון של תהליך ב'.

חסרון #2: מחסור בזיכרון רציף

- במערכת אמיתית, הזיכרון עובר קיטוע (fragmentation):

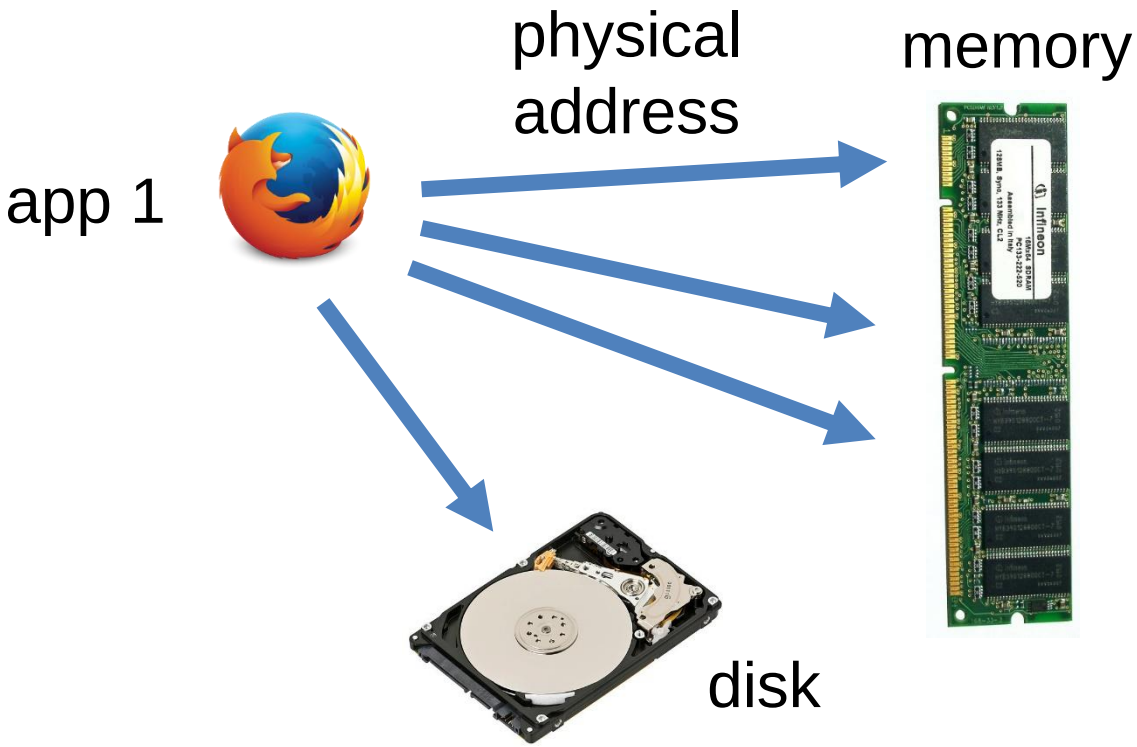


- גם כאשר יש מספיק זיכרון במערכת, הוא "שבור" לרסיסים.

פרגמנטציה (fragmentation)

- בזבוז זיכרון כתוצאה מאופן שימוש לא יעיל.
- יש שני סוגי פרגמנטציה:
- פרגמנטציה **חיצונית** (external fragmentation) – בזבוז **מחוץ** למקטעי הזיכרון בגלל שהם מפוזרים במרחב.
  - לדוגמה: מערך C חייב להיות מוקצה בצורה רציפה בזיכרון.
  - ייתכן כי לא נצליח להקצות מערך בגודל נתון למרות שיש מספיק זיכרון – סכום כל החורים במרחב גדול מספיק, אבל החורים לא מאורגנים באופן רציף.
- פרגמנטציה **פנימית** (internal fragmentation) – בזבוז **בתוך** מקטעי הזיכרון כתוצאה מהקצאת יתר.
  - לדוגמה: מהדר מסוים מיישר כל הקצאת זיכרון לכפולה של בלוק בגודל N.
  - אם המשתמש מבקש זיכרון בגודל $< N$, שאר הזיכרון יתבזבז.

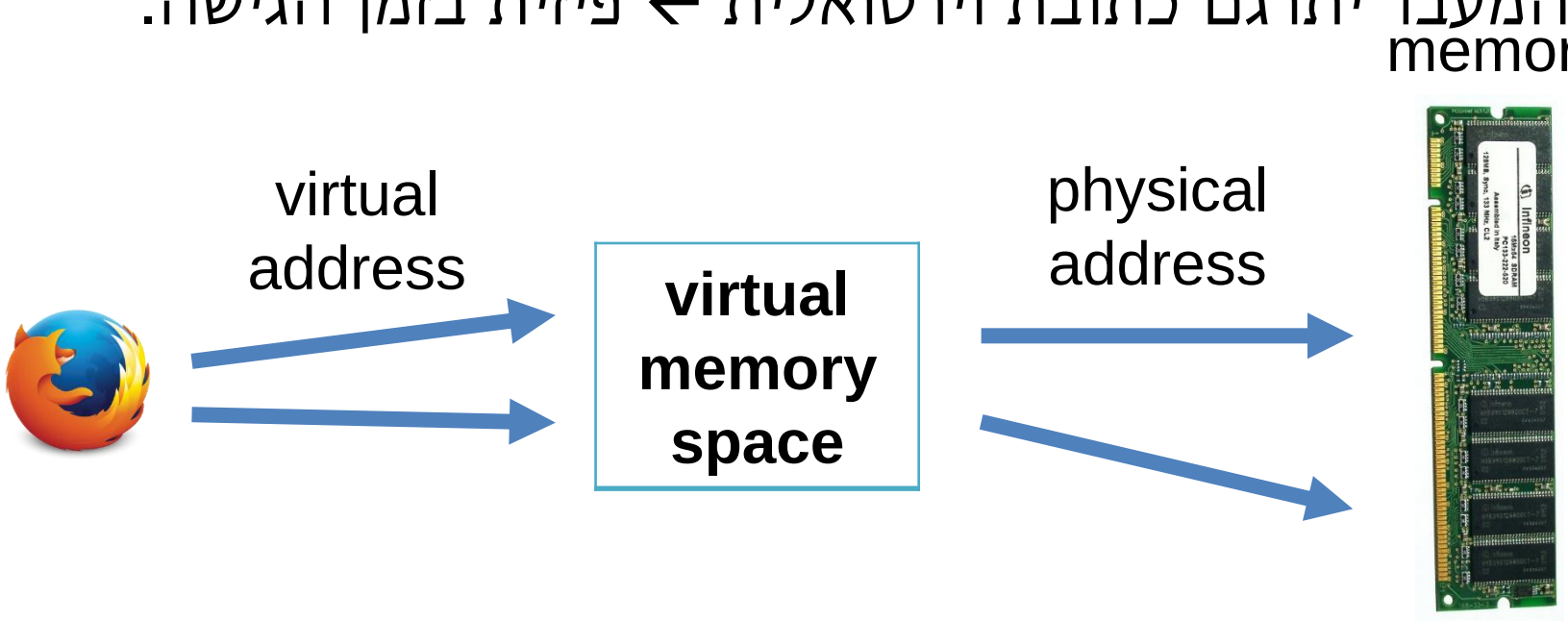
חסרון #3: מגבלת זיכרון



- היינו רוצים להשתמש גם בשטח האיחסון שקיים בדיסק, **בצורה שקופה לקוד האפליקציה.**

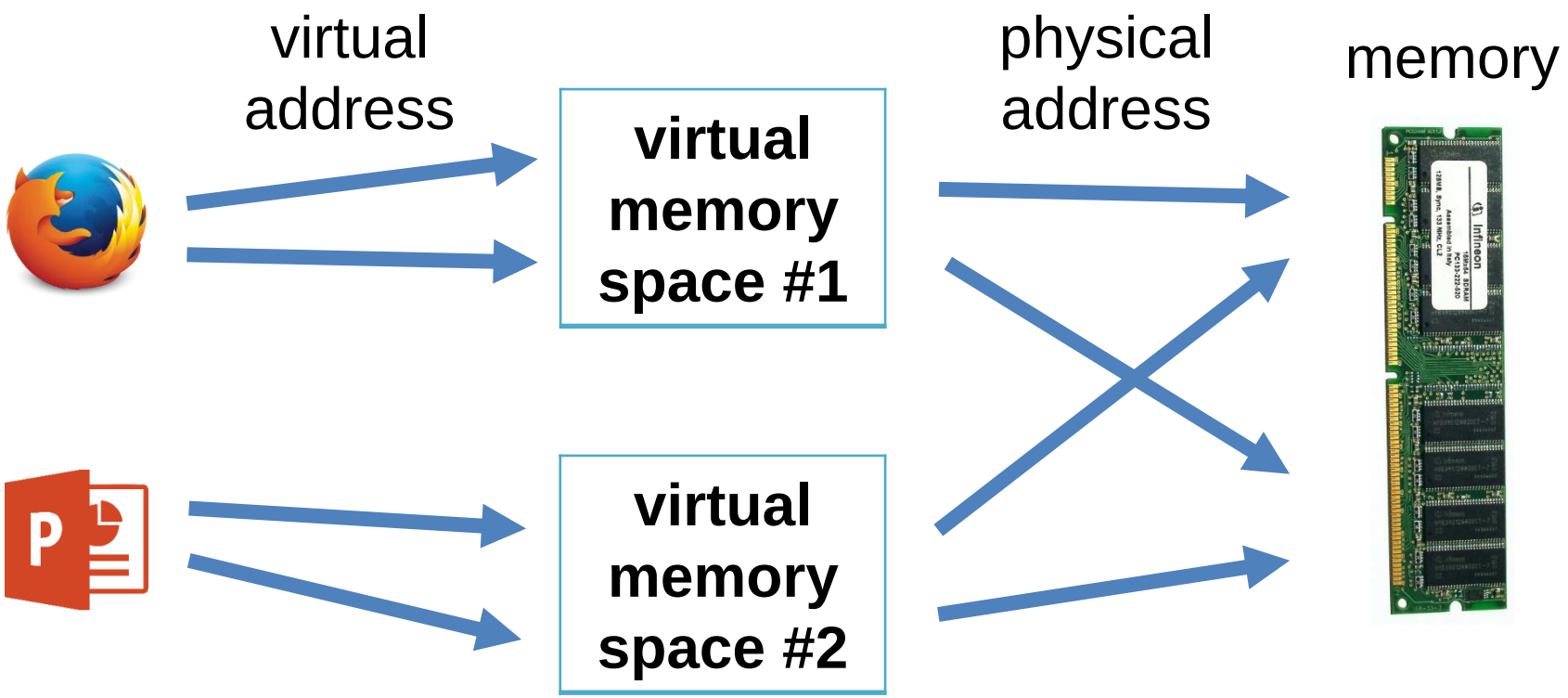
הפתרון: זיכרון וירטואלי

- פקודות המכונה ייגשו אך ורק לכתובות זיכרון וירטואליות.
- מערכת ההפעלה תגדיר מיפוי (== פונקציה) בין כתובות וירטואליות לפיזיות: לכל כתובת וירטואלית מתאימה בדיוק כתובת פיזית אחת.
- המעבד יתרגם כתובת וירטואלית ← פיזית בזמן הגישה.

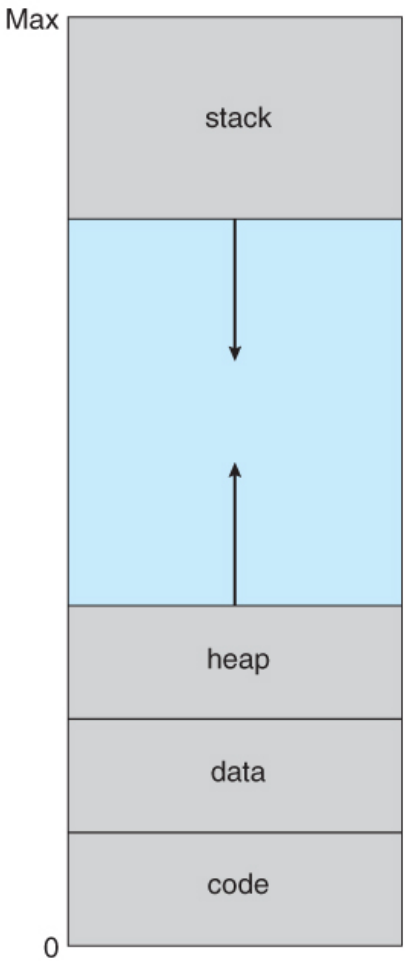


זיכרון וירטואלי נותן בידוד/הגנה

- תהליך יכול לגשת רק למרחב הזיכרון הוירטואלי שלו עצמו.
  - כל תהליך מקבל אשליה שהוא לבד במערכת.



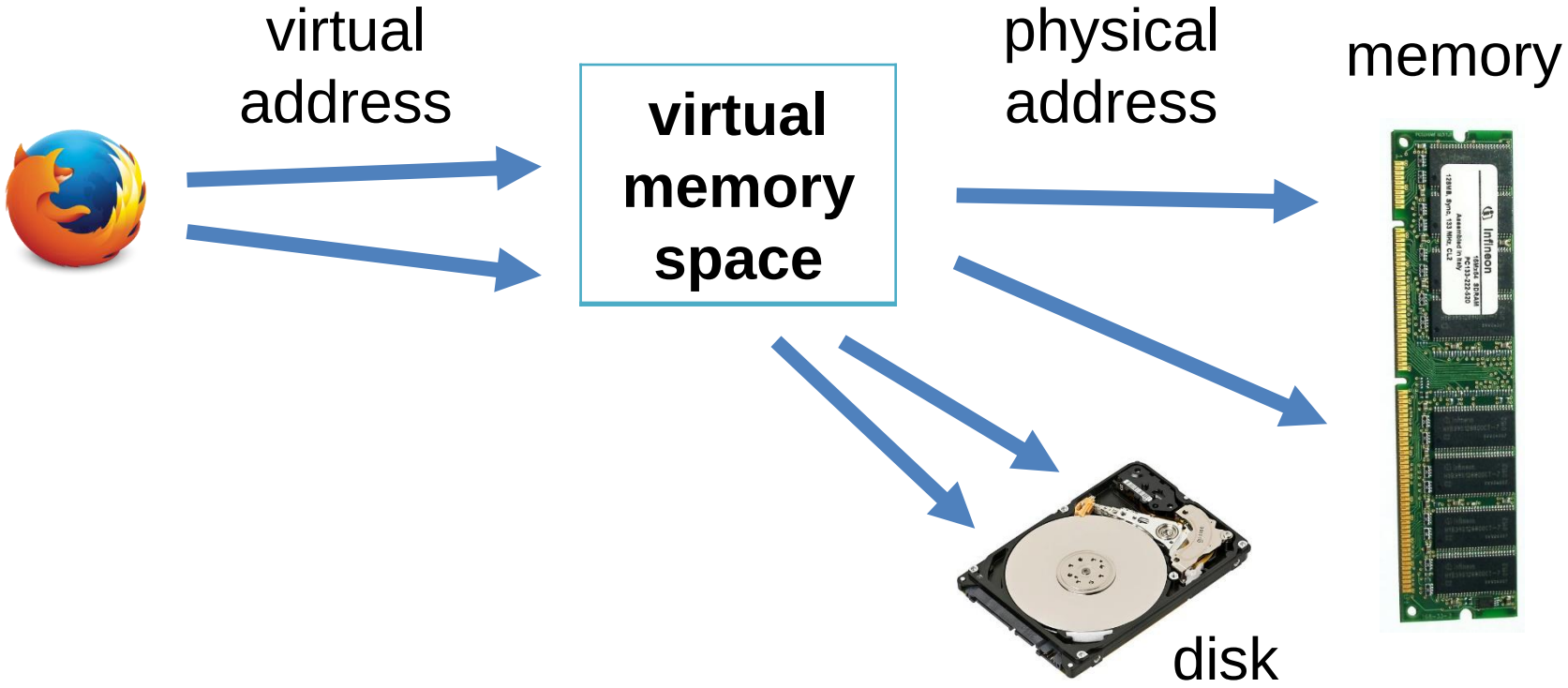
זיכרון וירטואלי מספק רציפות



- תהליך חדש מקבל מרחב זיכרון וירטואלי "נקי" ורציף.
- בנוסף, מרחב הזיכרון הוירטואלי של תהליך יכול להיות גדול הרבה יותר מהזיכרון הפיזי הזמין.
- ← מערכת ההפעלה תוכל למצוא בקלות יותר זיכרון רציף במרחב הוירטואלי.
  - שימו לב: הזיכרון הפיזי המתאים לא חייב להיות רציף!

זיכרון וירטואלי מאפשר swapping

- ניתן למפות חלקים מהזיכרון הווירטואלי אל הזיכרון או אל הדיסק.
- ← המשתמש יראה יותר זיכרון ממה שיש באמת במערכת.



זיכרון וירטואלי מציע יתרונות נוספים

- **demand paging** – חסכון של זיכרון פיזי ע"י הקצאתו רק בגישה הראשונה לזיכרון הוירטואלי.
  - למשל: אם הקצנו מערך גדול באמצעות malloc() ולא ניגשנו לחלקים ממנו, החלקים האלו לא יהיו מגובים בזיכרון הפיזי.
- **deduplication** – חסכון של זיכרון פיזי במידה ואפליקציות שונות משתמשות באותו מידע **לקריאה בלבד**.
  - למשל, מרבית התהליכים משתמשים במידע של ספריית libc (לקריאה בלבד).
  - לכל תהליך מרחב זיכרון וירטואלי שונה, אבל כולם יכולים למפות לאותו אזור פיזי שבו יושבת הספרייה libc.
- **copy-on-write** – מנגנון לחיסכון של זיכרון פיזי ולמניעת העתקות מידע מיותרות – נראה בתרגול הבא.
- ועוד יתרונות רבים אחרים...

PAGING במעבדי אינטל 32-ביט

או: איך מממשים זיכרון וירטואלי?

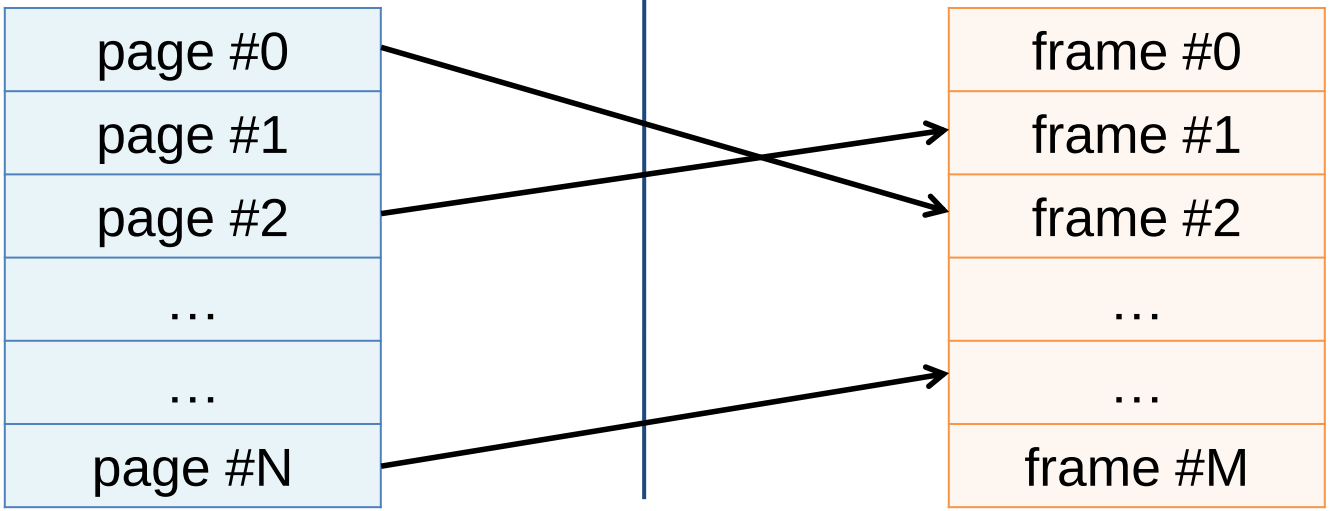
דפים ומסגרות

מרחב הזיכרון הפיזי

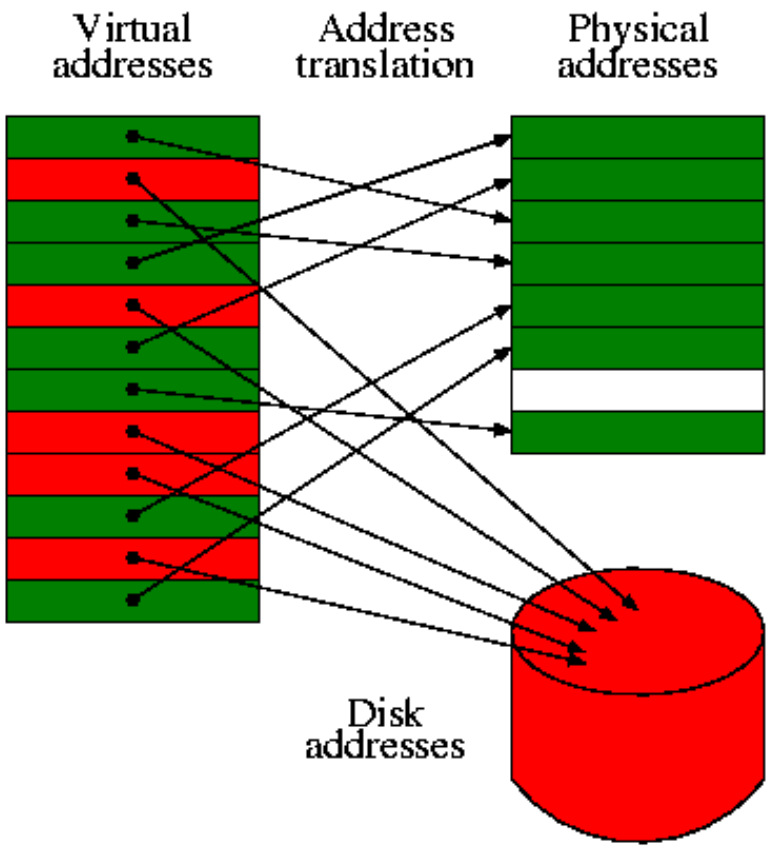
- מחולק ל**מסגרות** (**frames**) – בלוקים עוקבים בגודל קבוע (4KB בארכיטקטורת IA-32).
- המסגרות מיושרות בזיכרון הפיזי.

מרחב הזיכרון הוירטואלי

- מחולק ל**דפים** (**pages**).
- גודל דף == גודל מסגרת (4KB).
- הדפים מיושרים בזיכרון הווירטואלי.



לא כל הדפים ממופים למסגרות פיזיות!



- חלק מהדפים לא ממופים כלל, כלומר הדף לא מגובה בזיכרון ולא בדיסק.
  - מטעמי חיסכון בזיכרון ובזמן, אין טעם להקצות מראש את כל המרחב הווירטואלי של תהליך.
- חלק מהדפים יכולים להיות מגובים בדיסק.
  - נאמר כי הדפים swapped out.
  - פרטים נוספים בתרגול על מטמון הדפים.

דפים ומסגרות בארכיטקטורת IA-32

- בארכיטקטורת IA-32 (ארכיטקטורת 32 ביט של אינטל):
- הזיכרון **הווירטואלי** הוא ברוחב 32 ביט.
- הזיכרון **הפיזי** הוא ברוחב 32 ביט.

- מה מספר הדפים במרחב הווירטואלי?

$$\frac{space\ size}{page\ size} = \frac{2^{32}}{2^{12}} = \frac{4\text{GB}}{4\text{KB}} = 1\text{M} \cong 1{,}000{,}000$$

- מה מספר המסגרות במרחב הפיזי?
- כנ"ל (החישוב זהה).

מיפוי דפים למסגרות

- בכל גישה לזיכרון, המעבד מתרגם את הכתובת הוירטואלית לכתובת פיזית באופן הבא:

virtual address	
31 – 12	11 – 0
page number	page offset

המיפוי נשמר
בטבלת הדפים

ההיסט זהה
(אין מיפוי)

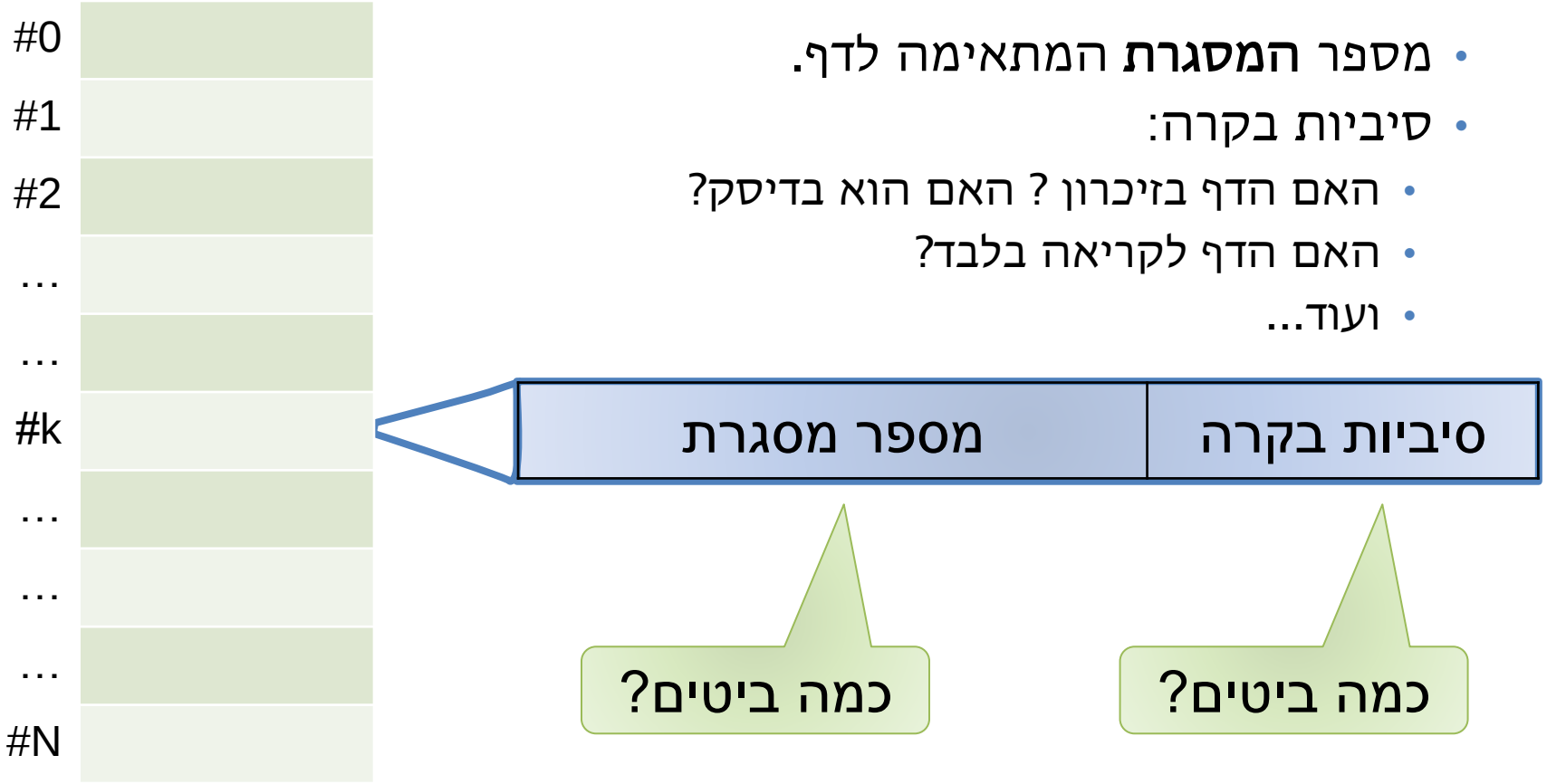
physical address	
31 – 12	11 – 0
frame number	frame offset

טבלת הדפים (page table)

- לכל תהליך יש טבלת דפים משלו – **מבנה נתונים אשר ממפה בין דפים למסגרות.**
  - ניתן לממש טבלת דפים באמצעות מבני נתונים שונים: מערך פשוט, עצים, טבלאות גיבוב (hash tables), ...
- עבור כל דף במרחב הזיכרון הווירטואלי של התהליך, יש כניסה בטבלת הדפים אשר מציינת:
  - האם הדף נמצא בזיכרון ובאיזו מסגרת?
  - האם הדף נמצא בדיסק ובאיזה מיקום?
  - האם הדף מעולם לא הוקצה? (כלומר איננו בזיכרון ואיננו בדיסק)
- טבלת הדפים אחראית לתפקידים נוספים כמו הגנת גישה.
  - למשל: טבלת הדפים מסמנת דפים לקריאה בלבד ומונעת גישות כתיבה.

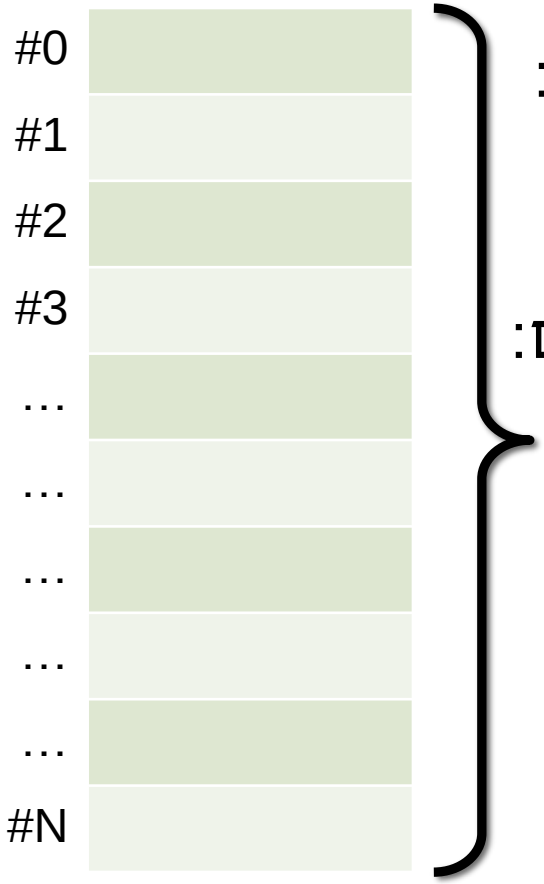
ניסיון #1: טבלת דפים ליניארית

- מערך שבו הכניסה ה-k מכילה את המיפוי של הדף ה-k.
- כל כניסה מכילה את:
  - מספר **המסגרת** המתאימה לדף.
  - סיביות בקרה:
    - האם הדף בזיכרון ? האם הוא בדיסק?
    - האם הדף לקריאה בלבד?
    - ועוד...



ניסיון #1: טבלת דפים ליניארית

- מה גודל טבלת הדפים?



- מספר הכניסות במערך הוא כמספר הדפים:
  N=1 M

- נניח שכל כניסה במערך היא בגודל **4** בתים:
  20 עבור מספר המסגרת
  + 12 עבור סיביות בקרה

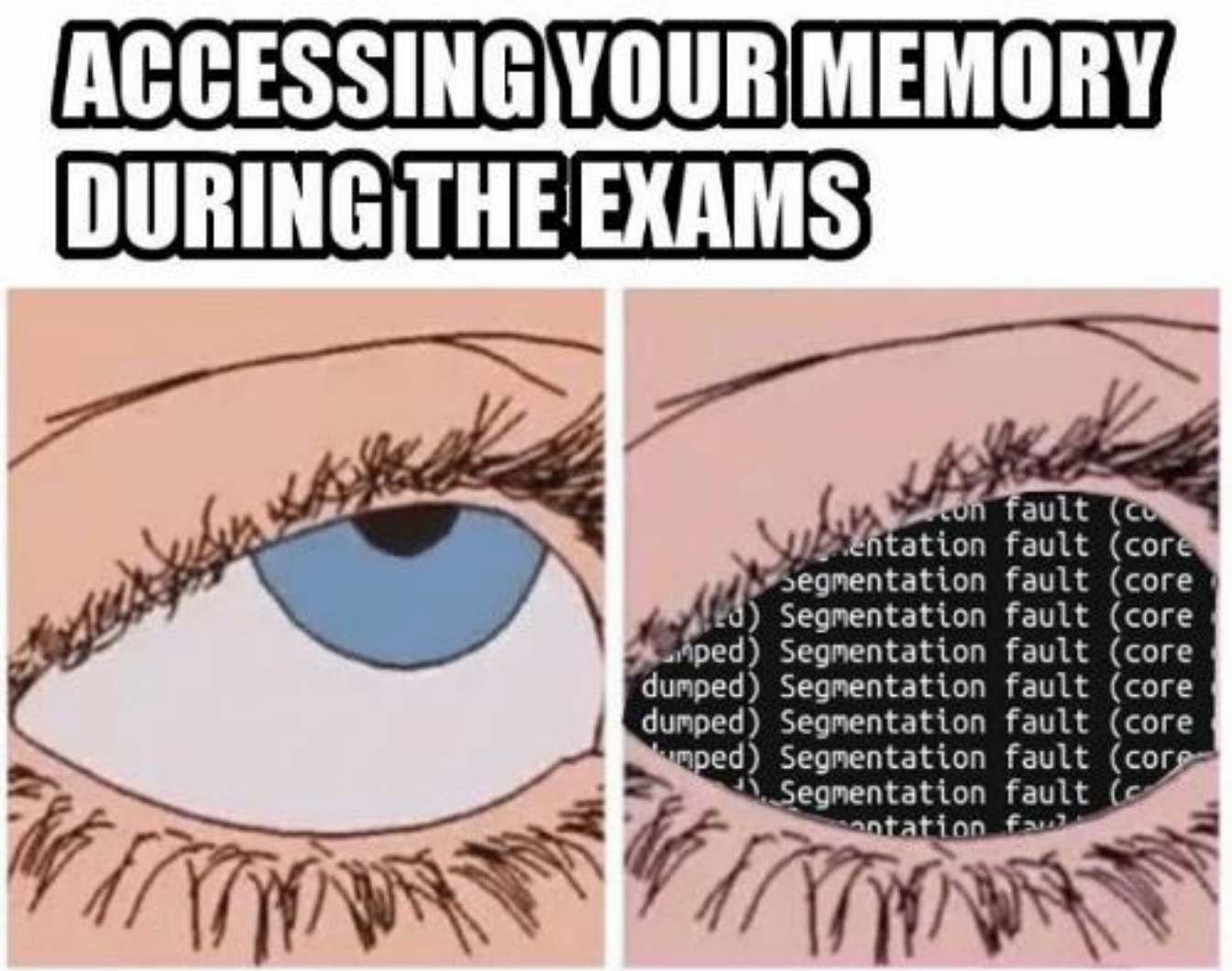
← גודל טבלת הדפים הוא 4MB
(ולכל תהליך טבלת דפים משלו!)

ניסיון #1: טבלת דפים ליניארית

- עבור 100 תהליכים, התקורה על הזיכרון הפיזי היא 400 MB.
- בפועל, תהליכים קטנים (ויש הרבה כאלו) ניגשים רק לחלק קטן של מרחב הזיכרון הווירטואלי, ולכן זה בזבזני להחזיק את הטבלה כולה.

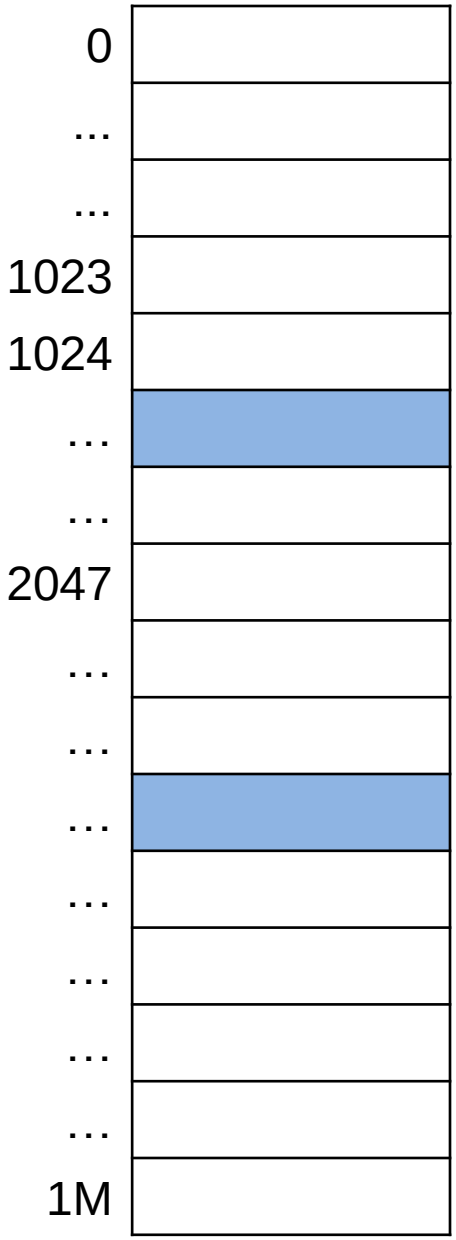
- יתרה מזאת, טבלת דפים ליניארית פשוט אינה ישימה בארכיטקטורות חדשות.
  - למשל, נניח כי רוחב של כתובת וירטואלית ופיזית הוא 48 ביטים.
  - מרחב הזיכרון הווירטואלי הוא בגודל 2^{48} B=256 TB.
  - גודל דף הוא 4KB ולכן יש $2^{48}/2^{12}=2^{36}$ כניסות במערך.
  - כל כניסה היא בגודל 8 בתים.
- ← גודל טבלת הדפים יהיה **512GB** לכל תהליך!

הפסקה

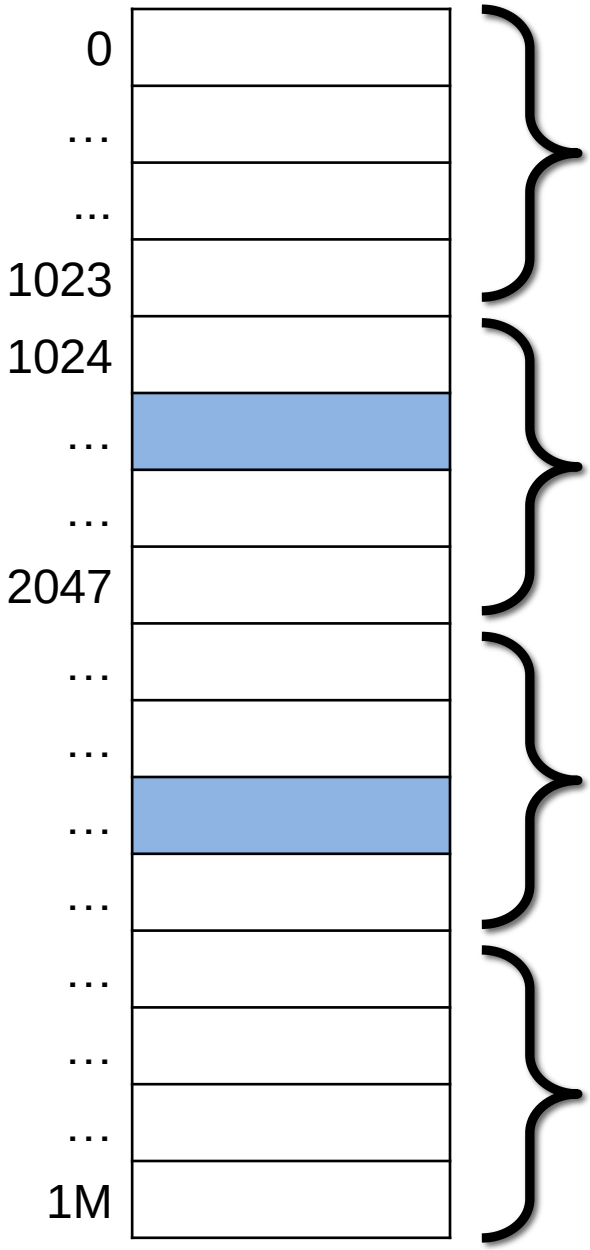

ACCESSING YOUR MEMORY
DURING THE EXAMS
Segmentation fault (core
Segmentation fault (core
Segmentation fault (core
Segmentation fault (core
Segmentation fault (core
Segmentation fault (core
Segmentation fault (core
Segmentation fault

ניסיון #2:
טבלת דפים היררכית

- נניח כי תהליך מסוים P ניגש לשני דפים בלבד.
- טבלת הדפים הליניארית של התהליך P משתמשת בשתי כניסות בלבד, כפי שמראה התרשים:
- קל לראות את בזבוז הזיכרון...

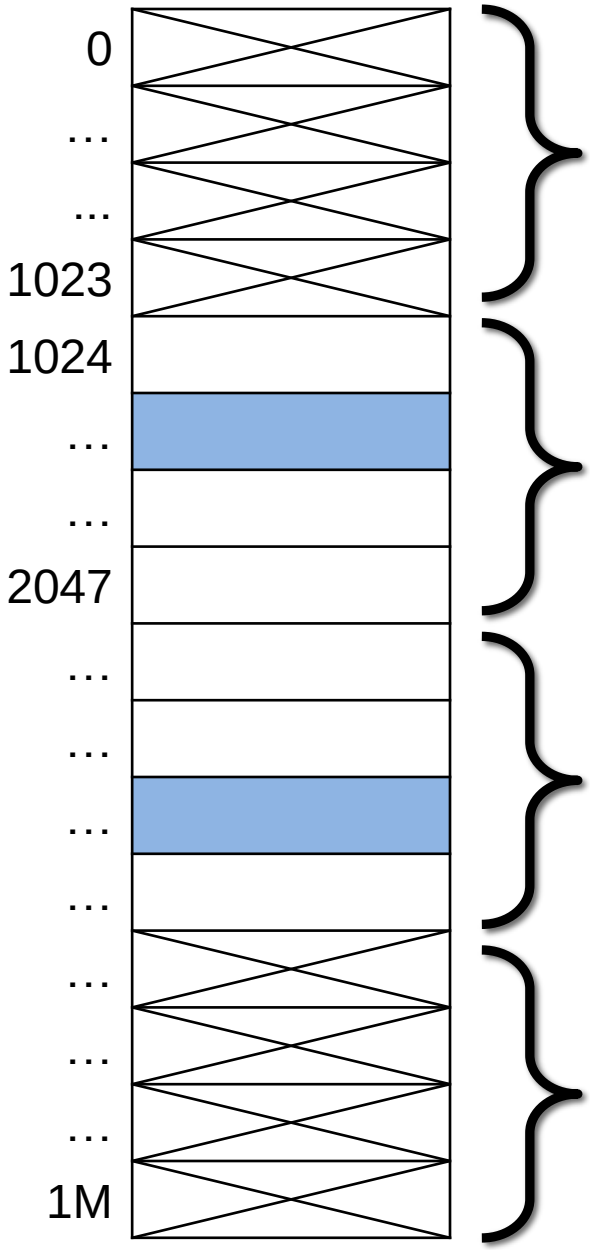


ניסיון #2:
טבלת דפים היררכית



- נחלק את הכניסות בטבלה לבלוקים בגודל מסגרת פיזית.
- כמה כניסות יהיו בכל בלוק?
- $\frac{frame\ size}{entry\ size} = \frac{4\text{KB}}{4\text{B}} = 1024$
- כמה בלוקים יהיו?
- $\frac{1\text{M}}{1024} = 1024$

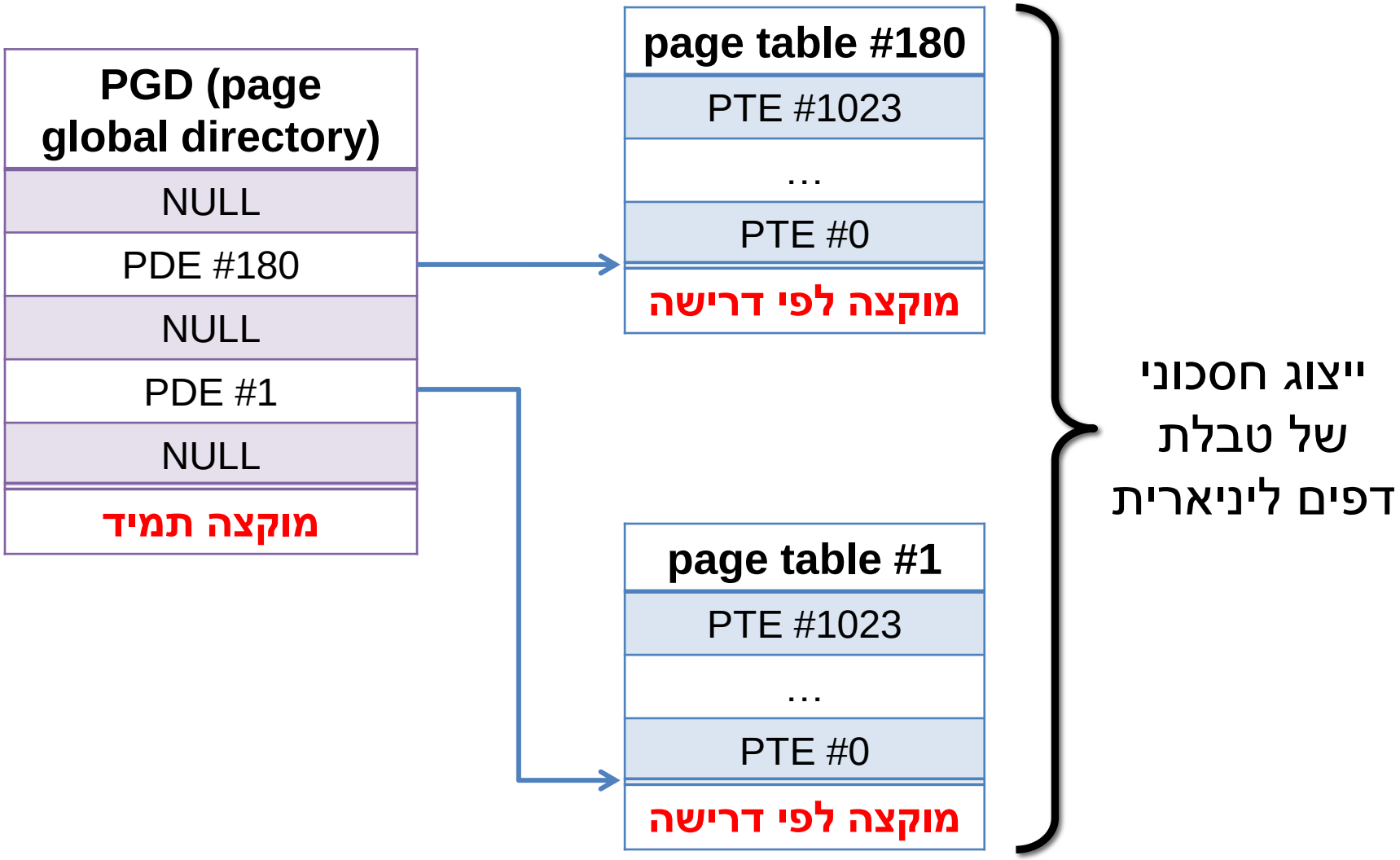
ניסיון #2:
טבלת דפים היררכית



- נשים לב כי מרבית הבלוקים ריקים לגמרי...
- ולכן לא כדאי להקצות אותם.

- נשמור טבלה נוספת עם 1024 כניסות אשר תצביע לבלוקים:
  - NULL עבור בלוק ריק.
  - כתובת פיזית עבור בלוק מוקצה.

ניסיון #2: טבלת דפים היררכית



תהליך תרגום כתובת וירטואלית

- איך המעבד מתרגם כתובת וירטואלית V לכתובת פיזית?
- בשלב הראשון, המעבד מחשב את מספר הדף הוירטואלי:

$$P = (V >> 12)$$

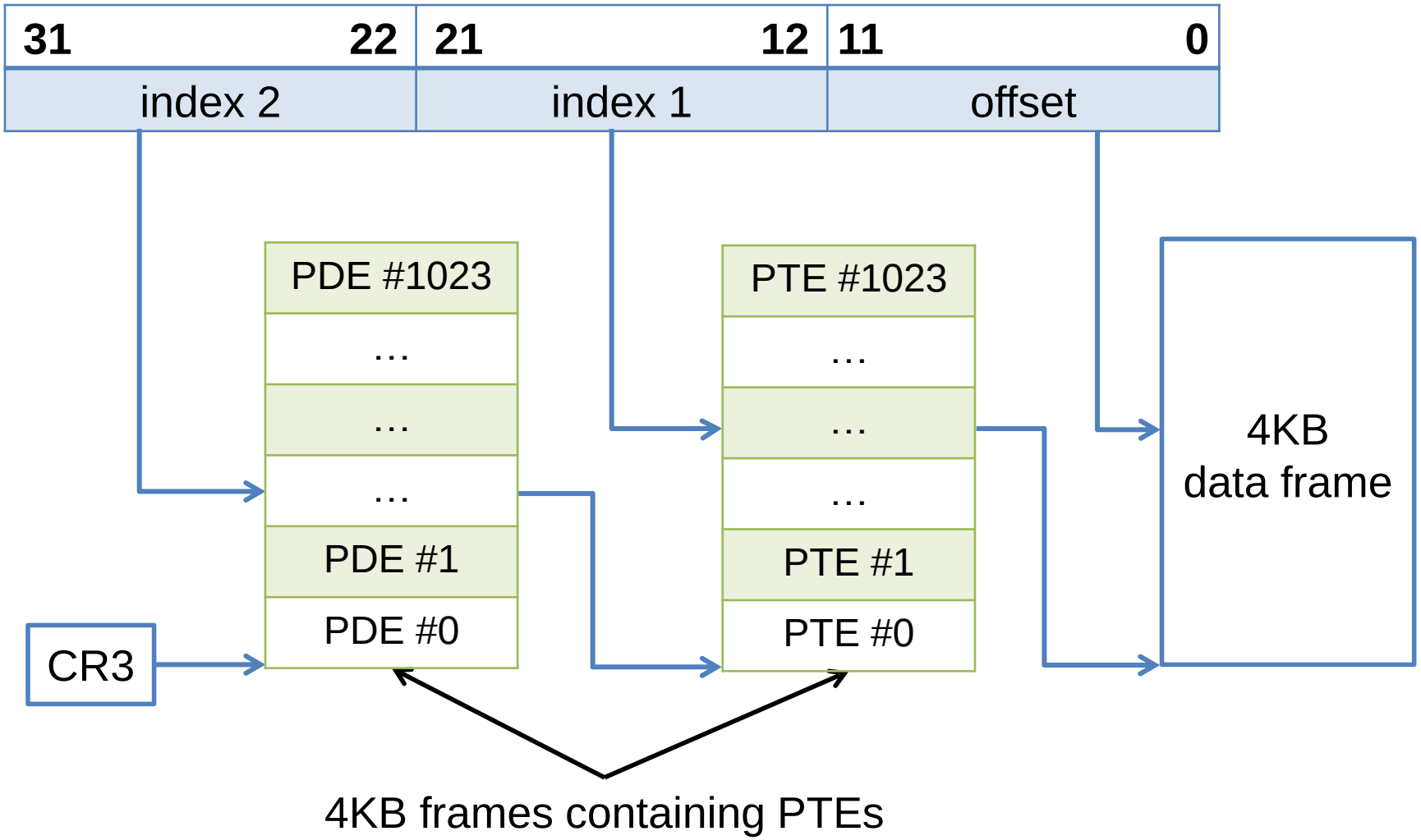
- <u>בטבלת דפים ליניארית:</u>

1. התרגום נמצא בכניסה **P** במערך.

- <u>בטבלת דפים היררכית:</u>

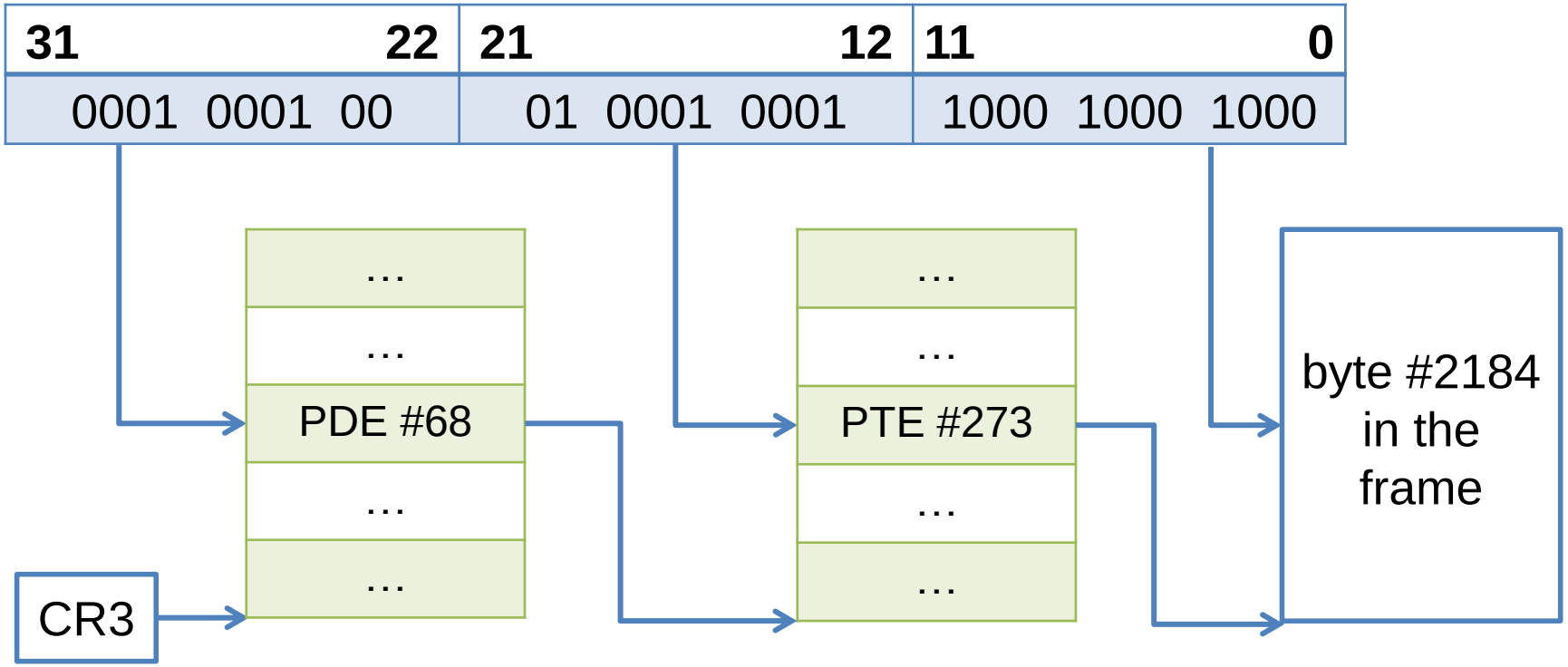
1. המעבד קורא את הכניסה **P / 1024** ברמה העליונה של העץ.
2. אם הכניסה הזו == NULL, אז אין תרגום (הדף לא בזיכרון).
3. אחרת, התרגום נמצא בכניסה **P % 1024** ברמה התחתונה של העץ.

פירוק כתובת וירטואלית לשדות



דוגמה: פירוק כתובת וירטואלית לשדות

- ניקח לדוגמה את הכתובת 0x11111888 .
- נכתוב את הכתובת בבסיס בינארי ונפרק אותה לשדות:



טבלת הדפים בארכיטקטורת IA-32

- בגלל הבעיות בטבלת הדפים הליניארית, אינטל בחרה בטבלת דפים היררכית בצורת עץ (דליל) עם שתי רמות.
  - מבנה הנתונים: radix tree במקום מערך.
  - הרמה התחתונה בעץ שומרת מיפויים בין דפים למסגרות---בדיוק כמו במערך.
  - הרמה העליונה בטבלת הדפים מצביעה למסגרות של הרמה התחתונה.
  - כל 1024 כניסות סמוכות ברמה התחתונה יישמרו במסגרת נפרדת (המסגרות לא בהכרח רציפות בזיכרון הפיזי, בניגוד למערך).
  - במידה ואף אחת מהכניסות ברמה התחתונה לא ממפה דף, אין צורך להקצות מסגרת ברמה התחתונה.
  - כאשר מוקצה דף חדש לשימוש התהליך, צריך להקצות, לפי הצורך, מסגרות עבור הרמות בהיררכיה עד (לא כולל) השורש.
  - רגיסטר מיוחד בשם **CR3** מצביע לשורש טבלת הדפים של התהליך הנוכחי.

האם CR3 מכיל כתובת וירטואלית או פיזית?

מבנה כניסה בטבלת הדפים

- כניסה ברמה הראשונה נקראת page directory = **PDE** entry.
- כניסה ברמה השניה נקראת page table entry = **PTE**.
  - בפועל, קוראים לכל הכניסות בכל הרמות PTE.
- כל כניסה בטבלת הדפים היא בגודל 32 bit.

- המידע שכניסה מכילה תלוי בביט **present** (ביט 0 של ה-PTE), המציין האם הדף נמצא בזיכרון הראשי.
- present == 1: הדף נמצא בזיכרון הפיזי.
- present == 0: הדף לא נמצא בזיכרון הפיזי.

Page Table Entry

G - Global
D - Dirty
A - Accessed
C - Cache Disabled
W - Write Through
U - User\Supervisor
R - Read\Write
P - Present

כניסה בטבלת הדפים, כאשר present==1

- **מספר המסגרת** בה מאוחסן הדף.
  - 20 ביטים, כאשר כתובות זיכרון פיזי באורך 32 ביט.
- **ביט accessed** (נקרא גם ביט referenced): מודלק ע"י החומרה בכל פעם שמתבצעת גישה לכתובת בדף. ביט זה מכובה באופן מחזורי ומשמש למדיניות פינוי הדפים לדיסק.
- **ביט dirty** (נקרא גם ביט modified): מודלק ע"י החומרה בכל פעם שמתבצעת כתיבה לנתון בדף. במידה והדף שייך לקובץ (לדוגמה) נידע שיש לכתוב אותו **חזרה לדיסק** מתישהו.
- **ביט read/write**: הרשאת גישה.
  - 0 = קריאה בלבד. 1 = קריאה וכתיבה.
- **ביט user/supervisor**: גישה מיוחסת.
  - 0 = גישה לקוד הגרעין בלבד. 1 = גישה לכל תהליך.

כניסה בטבלת הדפים, כאשר present==0

- יש שתי אפשרויות:

1. אם כל הביטים ב-PTE הם אפס, אז הדף לא ממופה כלל במרחב הזיכרון של התהליך.
2. אחרת, אם לפחות אחד מ-31 הביטים העליונים שונה מאפס, אז הדף נמצא במאגר דפדוף (swap area) בדיסק, וב-PTE נשמור את כתובתו ב-swap area.

TLB – Translation Lookaside Buffer

- תרגום כתובת וירטואלית לפיזית קורה כל גישה לזיכרון.
  - 30%--50% מהפקודות בתכנית ממוצעת ניגשות לזיכרון ← תקורה גבוהה.
- כדי לשפר את הביצועים, מעבדי אינטל מכילים מטמון (cache) מיוחד, ה-**TLB**, אשר מכיל את התרגומים האחרונים בהם השתמשו.
  - המעבד מחפש ב-TLB לפני החיפוש בטבלת הדפים. אם התרגום המבוקש נמצא ב-TLB, נחסכו גישות יקרות לזיכרון (כמספר הרמות בהיררכיה).
  - אם התרגום המבוקש לא נמצא ב-TLB, המעבד פונה לחפש בטבלת הדפים ואז מוסיף ל-TLB את התרגום החדש (לטובת הגישות הבאות לזיכרון).

page number	frame number	flags
12	25	r/w, accessed
55	93	accessed, dirty
…	…	

פסילת תוכן ה-TLB

- ה-TLB מכיל עותק חלקי של המידע הקיים בטבלת הדפים, ולכן מערכת ההפעלה אחראית לשמור על קוהרנטיות המידע ב-TLB.
- הגרעין חייב לפסול (invalidate) את תוכן ה-TLB במקרים מסוימים, לדוגמה:

1. כאשר הגרעין מוחק כניסה בטבלת הדפים (כדי לפנות מסגרת מהזיכרון לדיסק), הוא מוחק גם את הכניסה המתאימה ב-TLB.
   - אחרת, התהליך עלול לגשת למידע לא מעודכן, בגלל שה-TLB עדיין מצביע למסגרת שכבר פונתה מהזיכרון.
2. בעת החלפת הקשר, הגרעין מוחק את תוכן ה-TLB כולו.
   - התהליך הבא לביצוע ייגש למסגרות של התהליך שרץ לפניו.

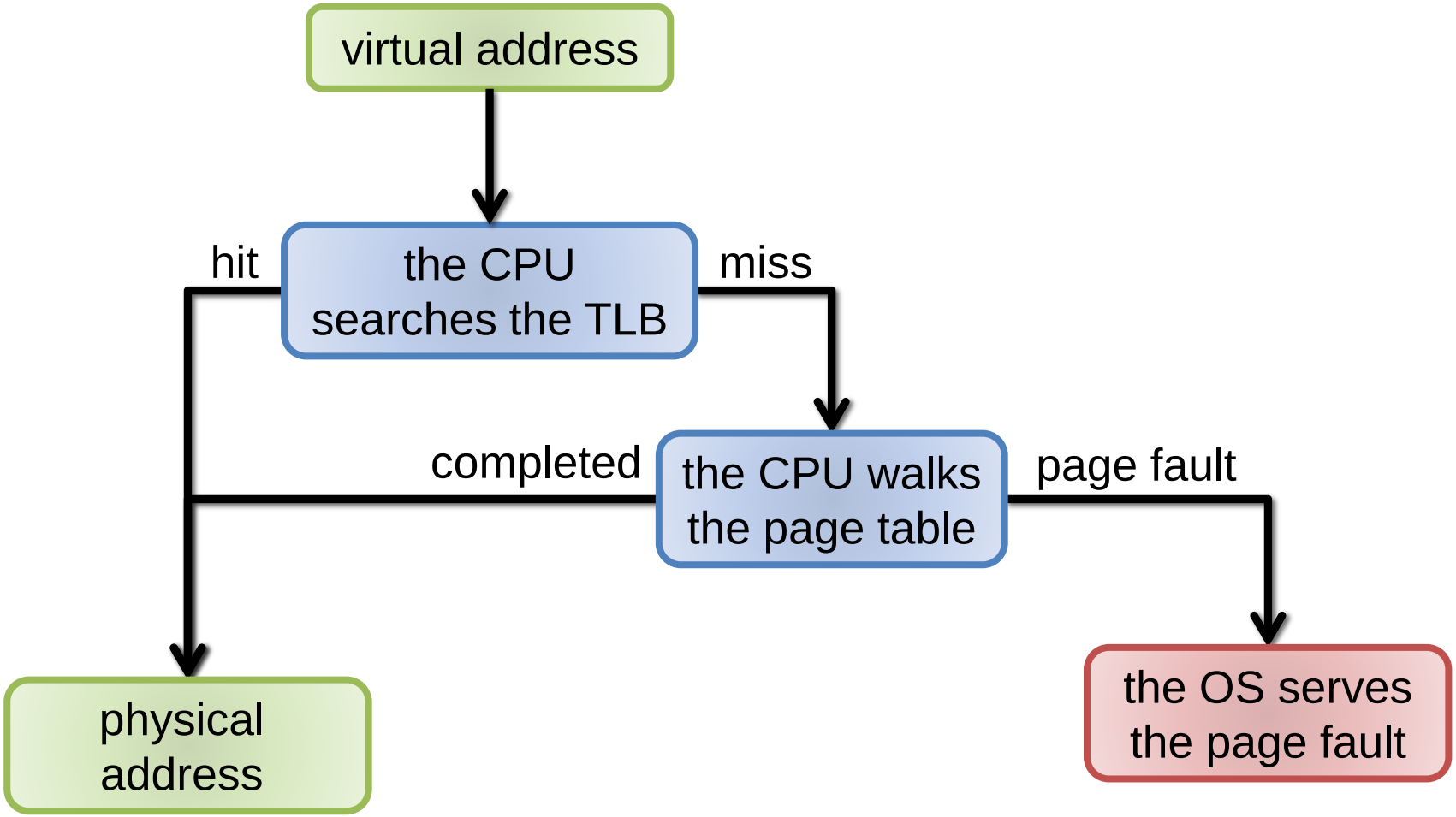
הימנעות מפסילת תוכן ה-TLB

- **שאלה:** למה כדאי להימנע מפסילת תוכן ה-TLB?

- לינוקס נמנעת מפסילת תוכן ה-TLB בהחלפת הקשר אם:

1. התהליך הבא לביצוע חולק את אותו מרחב זיכרון (אותן טבלאות דפים) יחד עם התהליך הקודם (שני חוטים של אותו תהליך).
2. התהליך הבא לביצוע הוא תהליך גרעין (**kernel thread**).
   - לתהליכי גרעין אין מרחב זיכרון משלהם, והם פועלים על מרחב הזיכרון של הגרעין.
   - תהליך גרעין מנצל את טבלאות הדפים של תהליך המשתמש שרץ לפניו, מפני שאין לו טבלאות דפים משלו.
   - **שאלה:** האם הגרעין יכול לגשת למרחב הזיכרון של התהליך הקודם?

סיכום: תהליך התרגום במעבדי אינטל



PAGING במעבדי אינטל 64-ביט

גודל מרחב הזיכרון

- במעבדי 64 ביט של אינטל (ארכיטקטורת x64), משתמשים בכתובות וירטואליות של 48 ביט בלבד (מתוך 64 אפשריים). מה גודל מרחב הזיכרון הווירטואלי?
- $2^{48}\ B = 256\ TB$

- מה היה גודל מרחב הזיכרון אם היו משתמשים בכל 64 הביטים לייצוג כתובות?
- $2^{64} = \mathbf{16\ Exabyte}$

- מדוע, אם כן, משתמשים רק ב–48 ביט?
- עבור האפליקציות הקיימות היום, אין צורך במרחב וירטואלי גדול כל כך.

אינדקסים לטבלת הדפים

- גם בארכיטקטורת x64 גודל הדף הסטנדרטי הוא 4KB.
- גודל כניסה בטבלת הדפים הוא 8 בתים.
  - ארכיטקטורת x64 תומכת בכתובות זיכרון פיזיות של עד **52** ביטים (שימו לב: מרחב הזיכרון הפיזי גדול יותר ממרחב הזיכרון הוירטואלי).
  - לכן מספר המסגרת מכיל $52 - 12 = 40\ bits$.
  - בתוספת 12 הביטים של הדגלים והרשאות הגישה יש לנו 52 ביטים.
  - גודל כניסה בטבלת הדפים הוא תמיד חזקה של 2, ולכן צריכים 64 ביטים, או 8 בתים.

- כמה כניסות (PTEs) מוכלות במסגרת (בגודל 4KB)?
- $\frac{4KB}{8B} = 512$

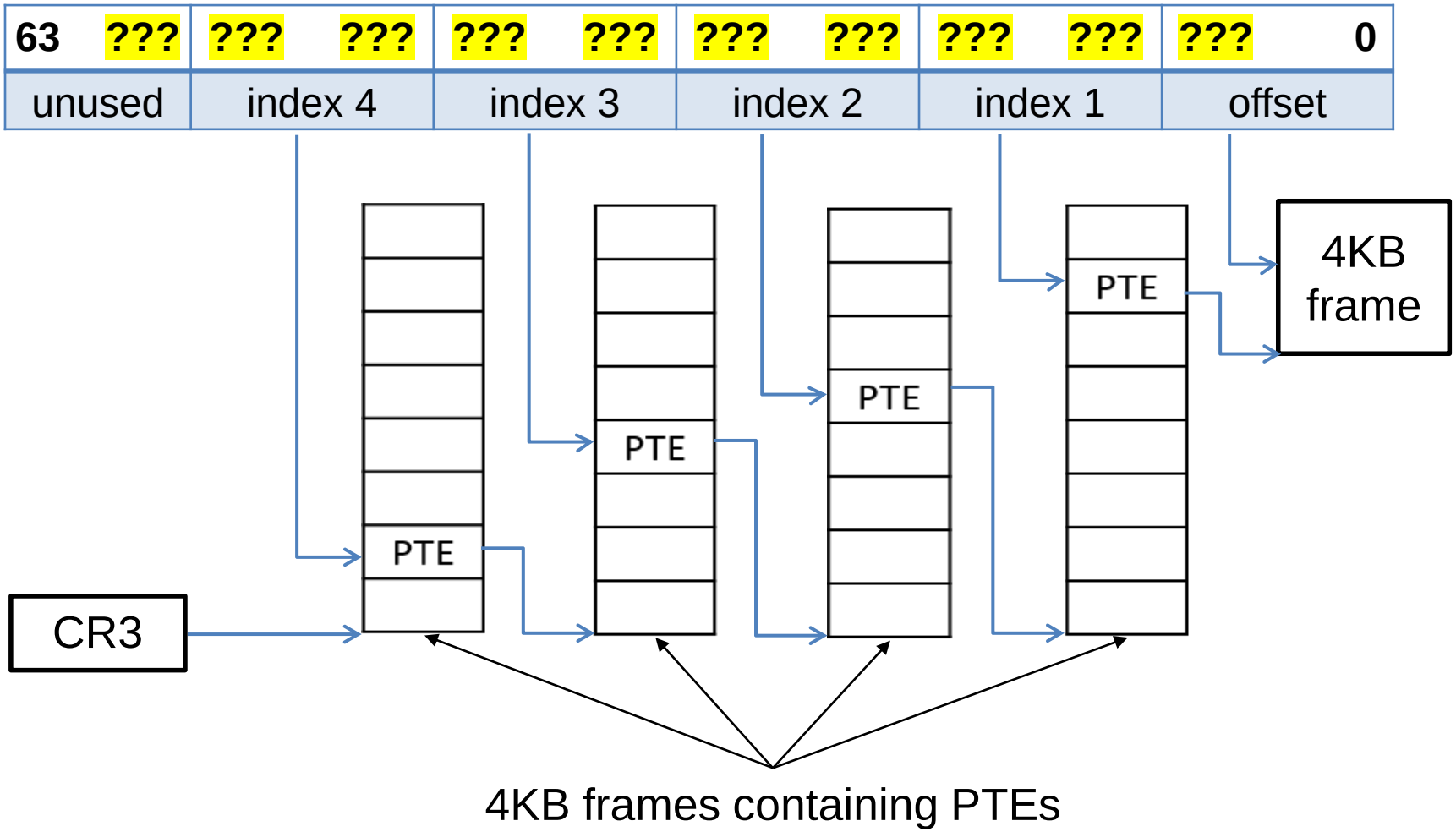
- כמה ביטים צריך כדי לאנדקס אותם?
- $\log_2 512 = 9$

page table walk

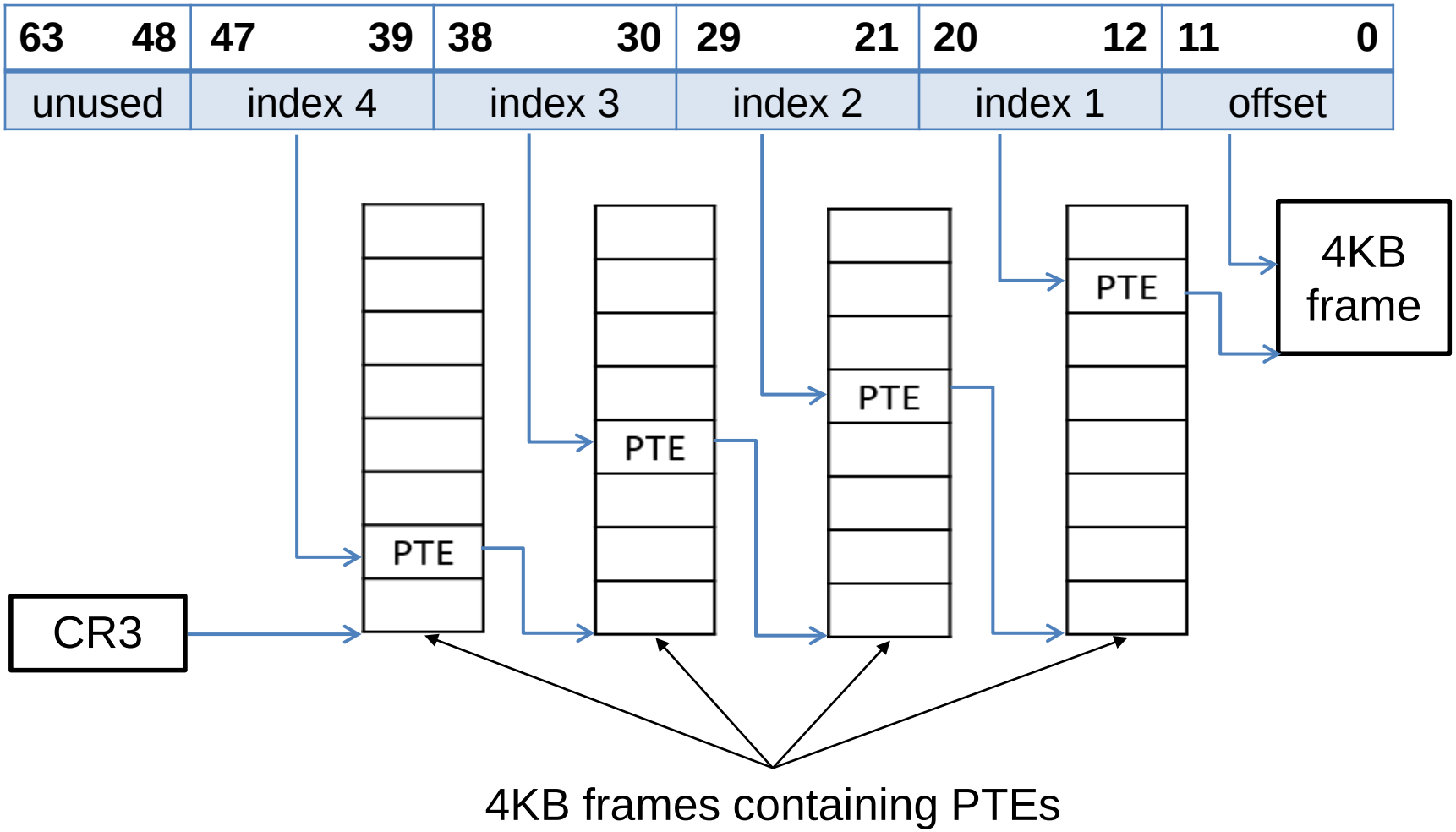
- מעבדי x64 משתמשים בטבלאות דפים עם ארבע רמות תרגום במקום שתיים.
- לצורך הפשטות, נקרא בשם PTE עבור כניסות בכל הרמות של הטבלה.
- גודל מסגרת בכל הרמות של טבלת הדפים הוא 4KB.

- תרגום כתובת וירטואלית לפיזית נקרא page table walk כי הוא "הולך" לאורך טבלת הדפים ההיררכית.
- השלימו את הסכימה הבאה:

page table walk



page table walk

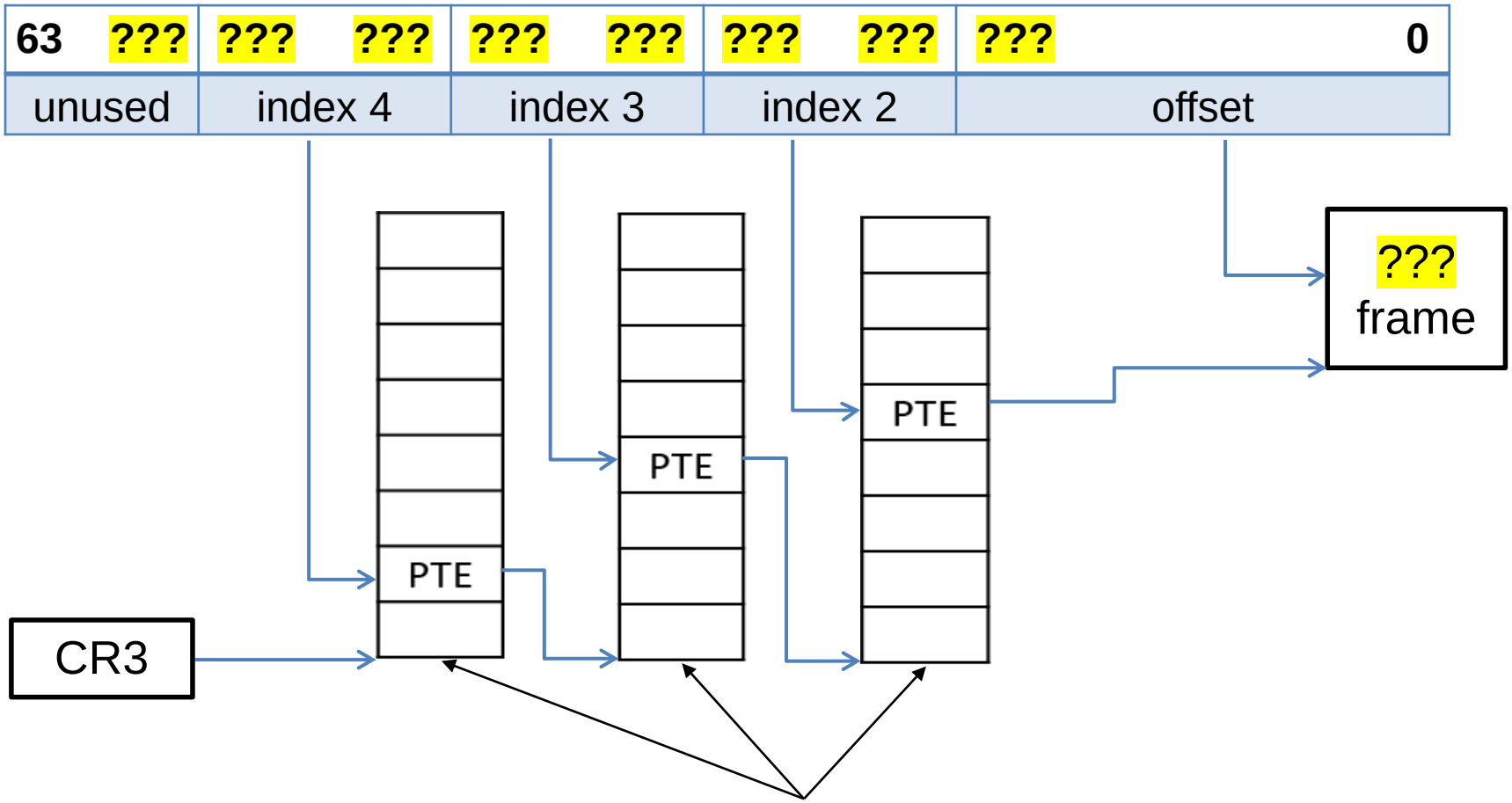


דפים גדולים

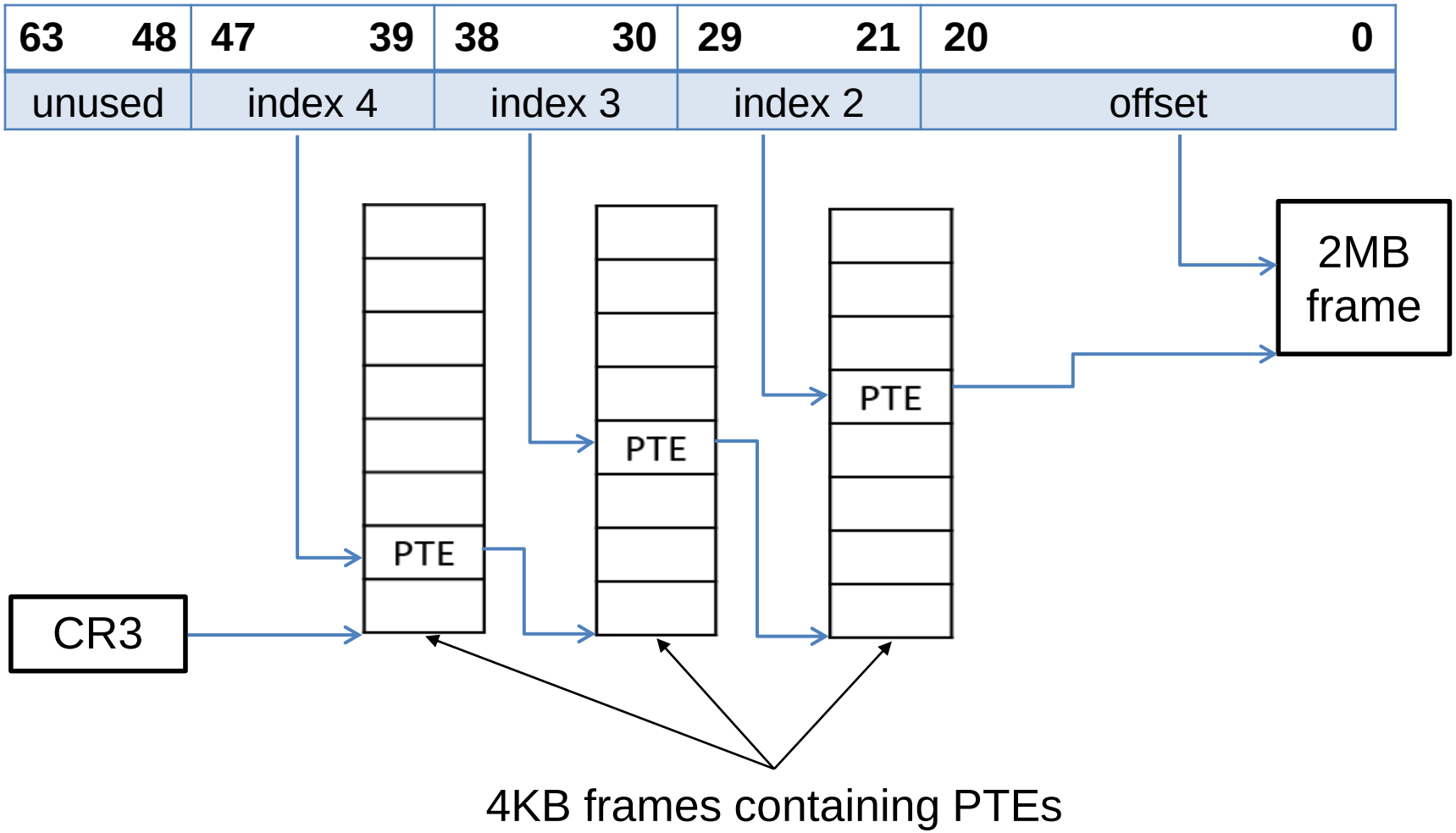
- מעבדי x64 תומכים גם בדפים גדולים (huge pages).
- תרגום של דפים גדולים "הולך" רק דרך 2 או 3 רמות של טבלת הדפים ההיררכית.
- הביטים של האינדקסים שנזרקו מצטרפים לשדה offset.

- השלימו את הסכימות הבאות ומצאו את גודל הדפים הגדולים:

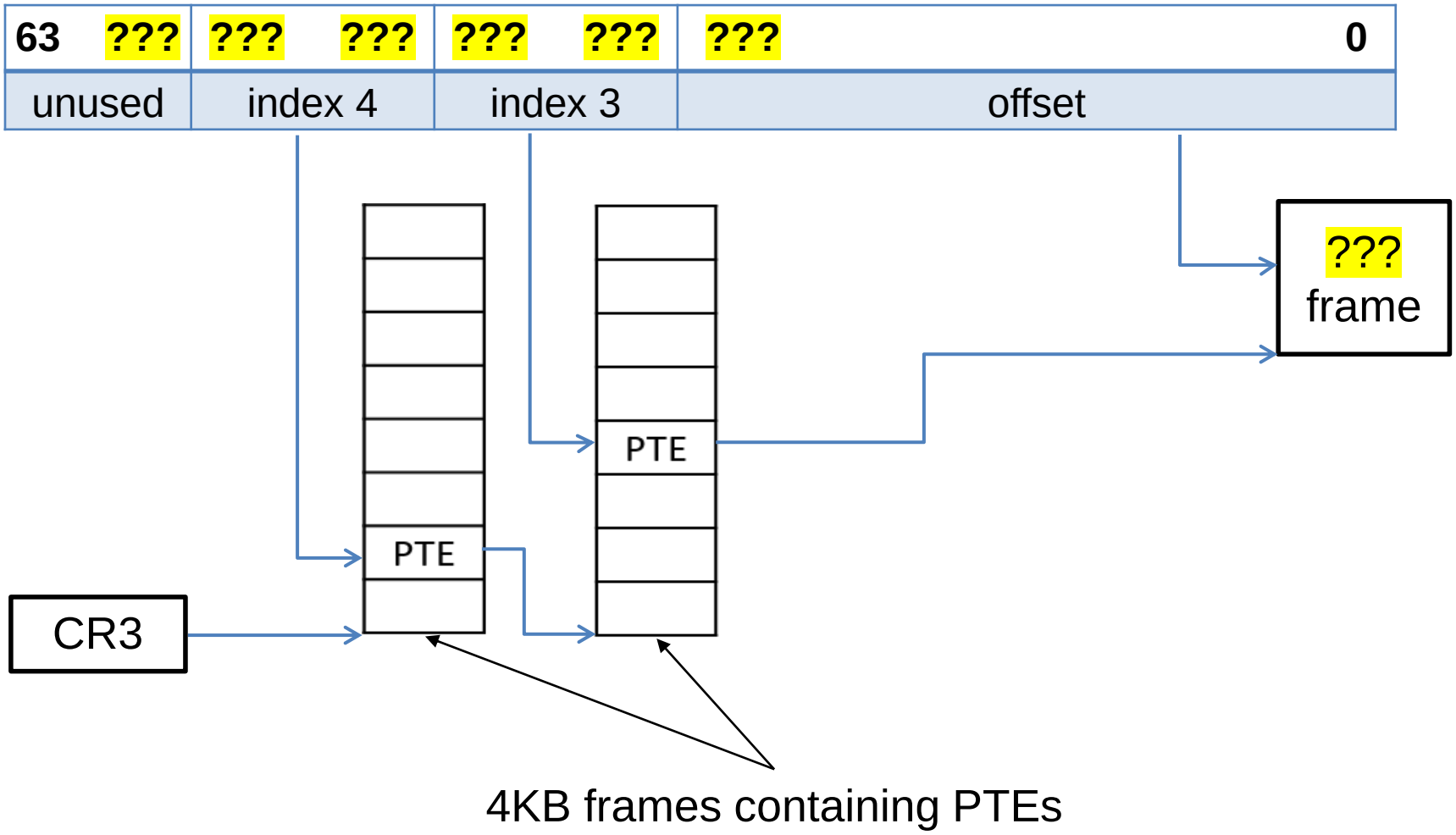
דפים גדולים


63 ??? ??? ??? ??? ??? ??? ??? ??? ??? 0
unused
index 4
index 3
index 2
offset
???
frame
PTE
PTE
PTE
CR3
4KB frames containing PTEs

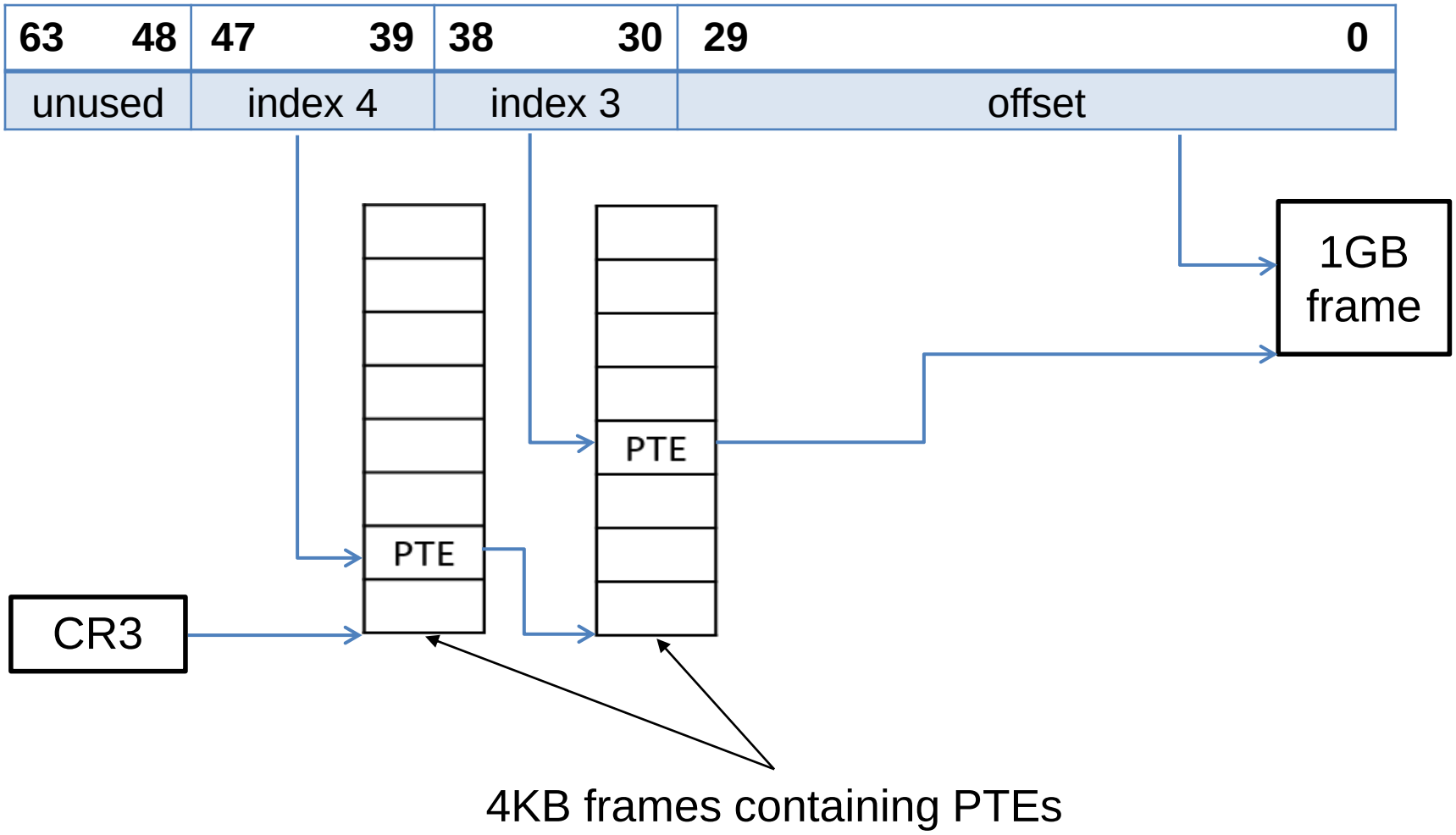
דפים גדולים


63 48
47 39
38 30
29 21
20 0
unused
index 4
index 3
index 2
offset
2MB frame
PTE
PTE
PTE
CR3
4KB frames containing PTEs

דפים גדולים


63 ??? ??? ??? ??? ??? ??? 0
unused
index 4
index 3
offset
???
frame
PTE
PTE
CR3
4KB frames containing PTEs

דפים גדולים


63 48
47 39
38 30
29 0
unused
index 4
index 3
offset
1GB frame
PTE
PTE
CR3
4KB frames containing PTEs

יתרונות וחסרונות דפים גדולים

חסרונות

- עלולים ליצור פרגמנטציה פנימית, כלומר לבזבז זיכרון בתוך הדפים.
  - לדוגמה: תהליכים קטנים ידרשו 2MB+2MB במקום רק 4KB+4KB למחסנית ולערימה.
- שימוש בדפים גדולים לצד דפים קטנים יוצר פרגמנטציה חיצונית, כלומר מחסור בזיכרון רציף.
  - מערכת ההפעלה תתקשה למצוא "חורים" לדפים הגדולים.
- פוגעים בביצועים (מבחינת latency).
  - לדוגמה: page fault יהיה ארוך יותר (יעתיק דף גדול יותר מהדיסק למטמון הדפים).

יתרונות

- משפרים ביצועים (מבחינת throughput) כי הם מקטינים את התקורה של תרגום כתובות.
  - מגדילים את יעילות ה-TLB (כל כניסה ב-TLB מכסה אזור זיכרון רחב יותר).
  - מקצרים את ה-page table walk.
  - בעקיפין, מגדילים את יעילות L1/L2/L3 caches (המטמונים שומרים את ה-PTEs, ועבור דפים גדולים יש פחות PTEs).