חזרה על השבוע שעבר:

מה המוטיבציה לזיכרון וירטואלי?

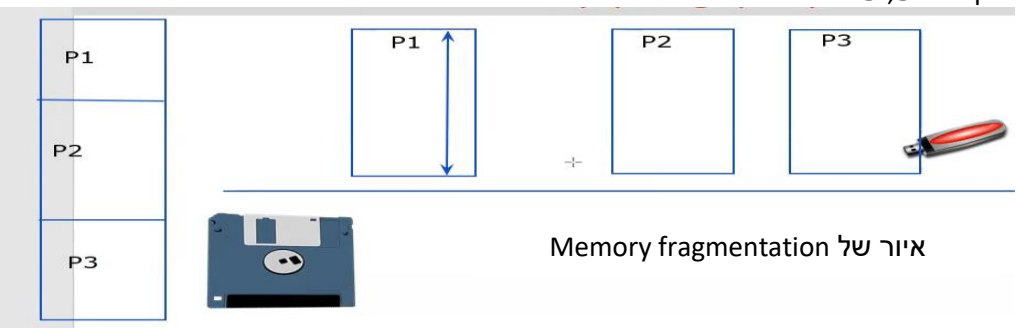
בהתחלה היה לנו CPU (=מעבד) עם ארכיטקטורות שונות, כך שה-RAM, memory chips שהיו שונים בטווחים שלהם בכך שהיו ממופים במעבד לכתובות פיזיות שונות.
מעבר של תוכנה ממחשב עם מעבד אחד, אל מחשב עם מעבד אחר היה נותן גודל של זיכרון פיזי שונה וגם כתובות פיזיות שונות, וזה בעייתי למפתח שפעם היה מפתח ב-commend line.
משום כך, כשמתכנת היה כותב קוד על מעבד מסויים, ובו הייתה התייחסות לכתובות פיזיות, כאשר היו מריצים את הקוד על מעבד אחר סביר להניח שהיו קורות תקלות.

הבינו שצריך API שיעטוף הקצאה של זיכרון ב API שייתן איזושהי קביעות למה שאנחנו עושים, לדוגמה ייתן כתובות שמצפים להן בערך.

הניסיון הראשון: base and offset
היה שנקבל כתובות פיזית ונוסיף לו offset כלשהו כדי להביא אותו לבסיס שנרצה נקרא לזה baseoffset שהיה תלוי בארכיטקטורה של מעבד וצ'יפים- אנחנו נקבל כתובות פיזיות מהמעבד שיהיו מגוונות.
אם נרצה למשל שכל הזיכרון יתחיל מ-0 אז ה-baseoffset היה שלילי ומניב כתובת וירטואלית שהיא 0.
על ידי זה שאנחנו יודעים חומרה היינו מביאים את כל הזיכרון לכתובת שנרצה, למשל 0, והיינו מקבלים כתובת וירטואלית שהוגדלה לפי virtAddr = addrPhys + NormilizationOffset
ואז הזיכרון שלנו מתחיל בטווח ידוע.

נניח שהחלטנו שכתובות וירטואליות היו מתחילות מכתובת 0 או 100,000, ואז היינו מקבלים - ללא תלות בהבנת איך החומרה עובדת- כתובות וירטואליות שמתחילות מאיפה שרצינו (למשל 0) ועד איפה שרצינו ועוד גודל הזיכרון שאנחנו צריכים (למשל נצטרך 120 כתובות בזיכרון, החלטנו שמתחיל מ-0 אז הכתובות הוירטואליות שלנו יהיו מ-0 ועד 119)

אבל אז הדרישות והמוטיבציות גדלו- הייתה הפרדה בין userSpace ל-kernalSpace, בעקבות כך החליטו להפריד בין תחום הכתובות של user מתחום כתובות של kernel.
בנוסף, הומצא מושג של ריבוי תהליכים. רצו לתת לכל תהליך איזשהו חלק שלו במעבד ע"י אלגוריתמים שונים. חוץ מהCPU לכל תהליך היה צורך בזיכרון משל עצמו, וכדי לספק את המענה של ריבוי תהליכים צריך היה לחשוב איך לתת זיכרון לתהליכים האלה.
בזמנו, RAM וזיכרון פיזי היה נורא יקר והיה מעט ממנו, לכן המחשבה ההתחלתית הייתה לקחת את הזיכרון הכללי להגדיר אותו כזיכרון וירטואלי ופשוט לתת חלקים קבועים לכל תהליך. זה אחד מה-design הראשונים של IBM.
הוא לא נתן מענה לתהליכים עם צרכים משתנים מבחינת זיכרון ולא נתן אפשרות לאלוקציה מתהליך שכבר לא צריך חלק מהזיכרון שלו, לתהליך אחר.
הזיכרון הפיזי היה מחולק בין התהליכים, ולכל תהליך היה תחום כתובות וירטואלי משל עצמו שהיה מכוסה בשלמותו עם כתובות פיזיות. סה"כ כל תהליך היה מקבל תחום כתובות וירטואלי בגודל שהוא הצהיר שהוא צריך מראש, שהיה ממופה כולו לכתובות פיזיות.



איור של Memory fragmentation

שיטה זאת לא הייתה רצויה כל כך מכיוון ש-CPU רוצה לבצע כמה שפחות אריתמטיקה על כתובות זיכרון.
(עבודה מיותרת עבורו) היו כמה שיטות ביניים שחשבו עליהן, בסוף בחרו לעבוד עם Pages. הרעיון מאחורי השיטה הזאת הוא ש-Page היא יחידת זיכרון קטנה, וכאשר תהליך רוצה זיכרון עבור אחד הצרכים שלו כמו

heap, stack, data... הוא מבקש Page וזה אמור להספיק לו באותו רגע. המוטיבציה ל-design הזה היא מכמה סיבות: א. Multi-Programming הוא דבר גמיש.
ב. חיסכון בזיכרון.
ג. קידם את הרעיון של מערכות משותפות גדולות.

לפי שיטת הדפים, כאשר אנו מקצים זיכרון וירטואלי בצורה של דפים אנחנו צריכים לדעת למפות בין הזיכרון הוירטואלי, לבין דף הזיכרון הפיזי. לצורך כך, קיימת טבלה בשם Page table שתפקידה להחזיק את המיפוי המבוקש.
שאלה: למה שב-Page-table יהיה יותר כתובות מה-physical?
תשובה: בגלל שפעם עבור CPU פעולה אריתמטית לא הייתה דבר קל, יותר קל ל-CPU להגדיל Stack\Heap כי זה רצוף. (גם היום שהפעולות האלה יותר קלות, עדיין נשארו עם המודל הזה כי הוא באמת יעיל יותר. קיים מודל נוסף שבשימוש ע"י Intel שמשתמש בסגמנטים, ז"א התהליך מקצה סגמנט זיכרון עבור כל צורך)

עוברים למצגת של Page tables של בן גוריון.

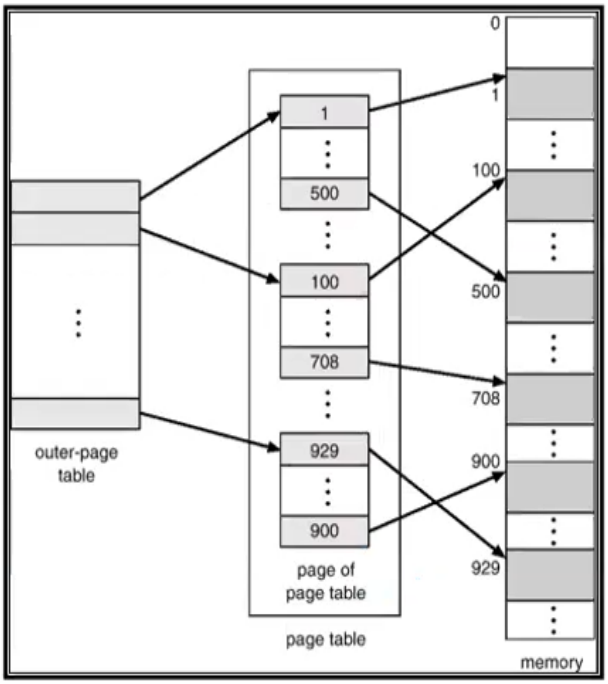
Page size VS. Page-table size
כתובת לוגית של 4 ג'יגה במערכת של 32-bit יכולה להתחלק ל: 1 קילו בייט דפים ו-4 מיליון רשומות, או 4 קילו בייט דפים ומיליון רשומות.
אינטואיציה: דף גדול -> כמות קטנה של דפים אבל עם פרגמנטציה גבוהה יותר.(מקום לא מנוצל בדף)
דף קטן -> כמות גדולה יותר של דפים ומכך טבלה גדולה יותר גם כן. (בזבוז של מקום)

ניתן לראות שמתקיים טרייד-אוף כאשר אנחנו בוחרים את גודל הדף.

Page 16K - 4 bytes/entry x 256 K entries = 1 Mb
Page 4K - 4 bytes/entry x 1M entries = 4 Mb
Page 1K 4 bytes/entry x 4M entries = 16 Mb

ניתן לראות בציור למעלה את הפערים כאשר בוחרים גודל דף קטן יותר. הפערים באין יותר לידי ביטוי במערכות מרובות תהליכים מכיוון שמה שאנו רואים למעלה מוקצה פר תהליך.

כיצד אנו מתמודדים עם טבלאות מיפוי גדולות מדי? קיצון אחד הוא תמיד להחזיק את טבלאות המיפוי בחומרה. גישה כזאת באופן גורף אינה אפשרית(יקר מדי), למרות שאנחנו כן נחזיק את חלק מהן על החומרה.(על ה-MMU) קיצון שני הוא להחזיק את הכל על הזיכרון הראשי. פתרון אפשרי הוא לשכלל את הטבלאות, נעשה אותן Multi-Level. פתרון אחד של Multi-Level הוא Two-Level Page-Table Scheme.



זוהי בעצם טבלה חיצונית שממפה לכתובת של כל טבלה פנימית ומשם מיפוי ישיר לכתובת הפיזית.
שאלה: איך החיסכון כאן לידי ביטוי?
תשובה: מכיוון שיש שני שלבים של מיפוי, **אז לא ייוצרו טבלאות ריקות כדי לשמור על הזיכרון רציף.**
אנחנו ניצור רק את המערך החיצוני רציף, ואז ניצור טבלאות חדשות רק בהתאם לצורך.

לסיכום, **חיסכון בזיכרון של Page-table עצמו.**

מס' הביטים שנשארו במערכת של 32-bit.
$32 - 12 = 20$

כי גודל הדף הוא $2^{12} = 4K$.

מצורפת דוגמא ל- Two level paging:

- A logical address (on 32-bit machine with 4K page size) is divided into:
  - a page number consisting of 20 bits.
  - a page offset consisting of 12 bits.
- Since the page table itself is paged, the page number is further divided into:
  - a 10-bit *p1* index.
  - a 10-bit *p2* index.
- Thus, a logical address has the following structure:

page number		page offset
p_1	p_2	d
10	10	12

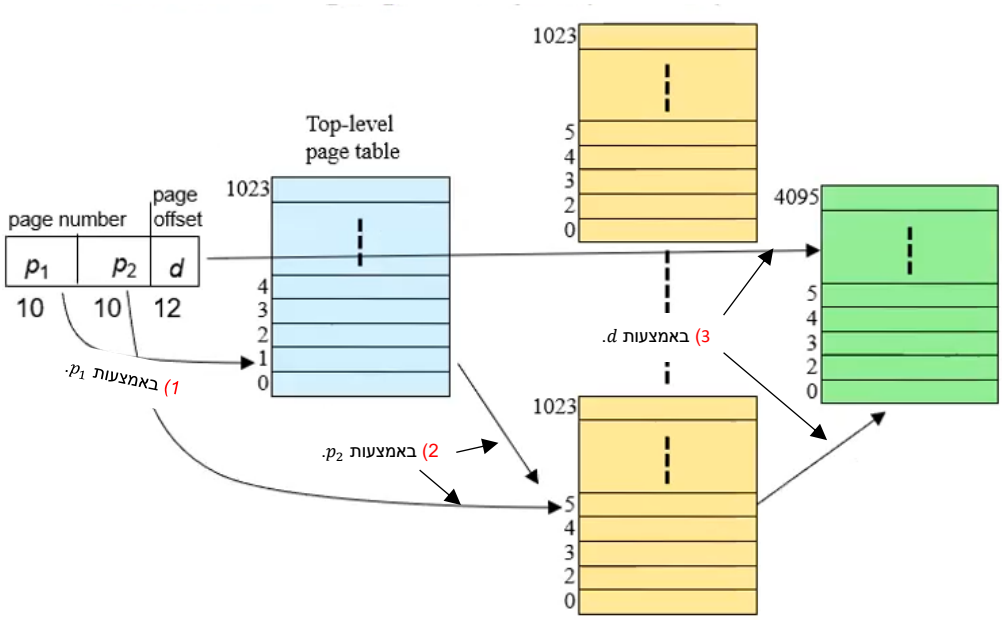
Where p_1 is an index into the top-level (outer) page table, and p_2 is an index into the selected second-level page table

דוגמא נוספת למוטיבציה מאחורי Two-level Paging:

- Two-level paging helps because most of the time a process does not need ALL of its virtual memory space.
- Example: A process in a 32bit machine uses
  - 4MB of stack
  - 4MB of code segment
  - 4MB of heap
- Only 12MB effectively used out of 4GB – only 3 page tables needed (out of 1024)

כל רכיב דורש 4 מגה, לכן יהיו לנו 1K דפים כאשר כל אחד מהם בגודל 4K. כמו כן, במערכת של 32-bit ע"מ לתאר דף צריך 4 bytes. לכן, ע"מ לתאר את כל הדפים של שלושת הרכיבים הללו צריך:
$3 \cdot \left(4\frac{bytes}{entry} \cdot 1K\ pages\right) = 12Kb$. ניתן לראות שהצלחנו למפות 12 Mb ל-12Kb.
(לפעמים ה-data לא רציף ולכן יכול להיות שנצטרך יותר דפים, אבל עדיין זניח.(יחסית)

דוגמא למיפוי Two-Level:



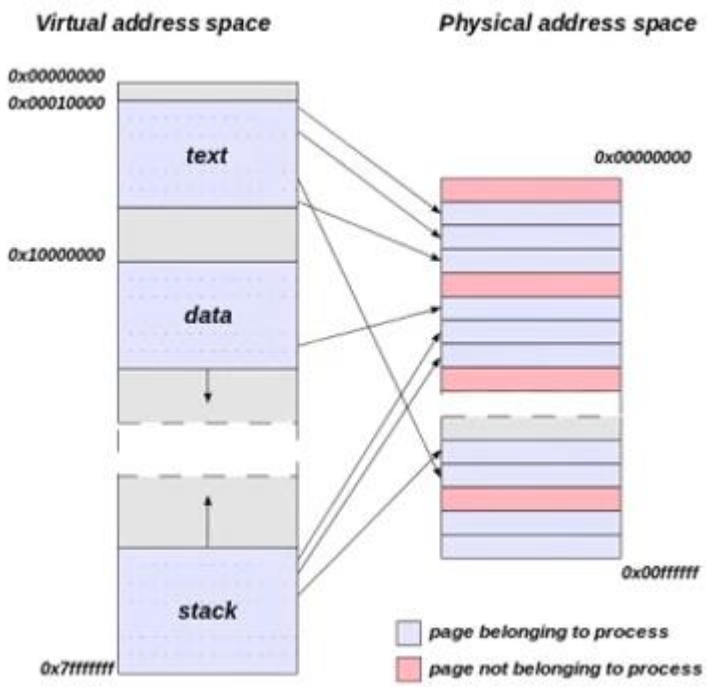
ע"מ למנוע מצב של חיפושים במקומות שכבר תופסים אנחנו נתחזק **TLB שזה סוג של מערכת caching.**

:Inverted Page-Table

Page table רגיל אנחנו ממפים את הכתובות הווירטואליות שיצרנו אל הכתובות הפיזיות שעל המחשב. כאן אנחנו מבצעים מיפוי הפוך כאשר אנחנו ממפים את הכתובות הפיזיות אל הכתובות הווירטואליות. במערכות של 64-bit אנחנו יכולים להגיד למצב שעבור 1-Gb של RAM וגודל דף של 4K כאשר כל רשומה בו היא בגודל 256K הטבלה תהיה בגודל של 2 MB! בנוסף, בשונה מהשיטה הקודמת כאן הטבלה היא עבור כל התהליכים.

הערה: אם הטבלה משותפת לכל התהליכים, ייתכן מצב ששני תהליכים ממפים בטבלה לאותו קטע קוד.(רצים על אותה תוכנית למשל) כדי שכל תהליך מיפוי יידע מי מבצע את השאילתה הזאת, אנחנו באים מהתהליך אל הטבלה עם שני משתנים: Pid, Virtual_Address. כמו כן, ברגע שנבצע חיפוש ונמצא מיפוי כלשהו, אז גם כאן נתחזק TLB שיהיה סוג של caching אלא שכאן הוא גם יהיה משותף לכל התהליכים.

הערה: באופן כללי, שנעבוד עם זיכרון ומיפויים תמיד נרצה להחזיק caching בגלל העיקרון של "**Locality of references**" שלפיו כתובות שפעם היו בשימוש, מאוחר יותר חוזרות.(בגלל גדילה ודעיכה של Heap\Stack)



נניח ויש לנו כתובות בזיכרון שכבר ממופות לדף כלשהו. אמרנו שניתן למפות את אותה כתובת לכמה תהליכים, אז נשאלת השאלה כיצד זה יתבצע? התשובה די פשוטה אומרת שעלינו לתחזק קאונטר לדף וכל פעם שמתווסף תהליך שמשתמש במיפוי הזה אנחנו מגדילים את הקאונטר המתאים.

Question 1: page table's size

- 64-bit computer
- Size of physical memory: 4GB
- Size of page: 4KB
- How many pages are possible?
- Each entry in the page table contains
  - Location of the required page frame in the physical memory (if presents)
  - 6-bits ctrl info (Valid, dirty, referenced etc).
- What's the size of the page table?

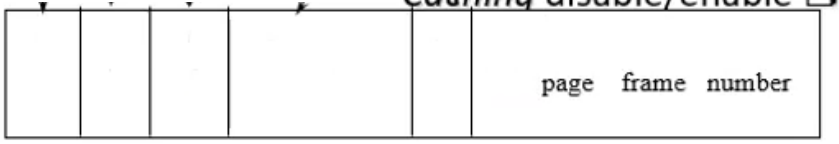
רוברט לא עבר עליה.

אמרנו שכל טבלה מורכבת מרשומות שממפות לכתובת בזיכרון. רשומה מוגדרת באיור הבא:

PTE

Page Table Entries (PTE) contain (per page):

- Page frame number (physical address) — מזהה של הפריים
- Present/absent (valid)bit — האם הוא בזיכרון של תהליך
- Dirty (modified) bit — האם הזיכרון שממופה לדף הזה עבר שינוי
- Referenced (accessed) bit — נותן אינדיקציה האם הדף הזה בשימוש
- Protection — רמת גישה. (read only\write\exec...)
- Caching disable/enable — האם הדף הזה יכול להיכנס ל-cache.(אם רוצים ביצועים גבוהים. או מידע רגיש)



TLB: (לפי רוברט חשוב מאוד- שווה את ההרצאה כולה)

.TLB- Translation Lookaside Buffer

בהתחלה היה CPU אבל לא מתוחכם כל כך, לכן עשו הכול בתוכנה. אבל אם לאחר חיפוש (LookUp) בטבלה היו מוצאים מיפוי כלשהו, היו מכניסים אותו לטבלה שתזכור אותו. לאחר תקופה יצרני מחשבים מצאו פיתרון חדש וטוב יותר והוא ה- MMU.(Memory Management Unit) מה עשה ה-MMU? ב-MMU הכניסו רשומות של TLB. ז"א ברגע שמצאנו מיפוי כלשהו נכניס ל-TLB. בתחילת הדרך היה TLB פרימיטיבי, ז"א היה לו רשומה כללית שהוא קיבל. בנוסף, לכל תהליך יש TLB משל עצמו שמבוסס תוכנה. עכשיו בסביבות מרובות תהליכים קורה הדבר הבא:

בהתחלה, ה-MMU מחזיק את ה-TLB של התהליך אליו שייך ה-Thread שרץ כרגע. באיזשהו שלב ה-Schedular מעלה Thread אחר, ומתרחש Context-Switch. מכיוון שהמיפוי זיכרון של התהליך הקודם עדיין נמצא ב-TLB שעל ה-MMU, צריך לבצע Flush ל-TLB הנוכחי שעל ה-MMU ולטעון את ה- TLBשל התהליך החדש.

שאלה של ראיון עבודה: מה ההבדל בין Context-Switch של Thread's מאותו תהליך לבין Context-Switch של Thread's מתהליכים שונים?

תשובה: ברגע ש-TLB מומש עם רשומה אחת מתרחשת החלפה (Flush) בין הרשומות על ה-MMU.

כאמור, עד עכשיו אלה היו פתרונות ישנים יותר. כיום המעבדים חזקים יותר ולכן ה-MMU יכול להכיל כמה רשומות TLB (1024) שמובדלות אחת מהשנייה ע"י Pid.

שאלת Follow-up: מה ההבדל בין Context-Switch של Thread's מאותו תהליך לבין Context-Switch של Thread's מתהליכים שונים כאשר המימוש של TLB מאפשר יותר מרשומה אחת?

תשובה: אין הרבה הבדל (לפחות ב-1024 הראשונים) כי אין צורך לעשות כבר Flush שרוצים להכניס רשומה חדשה.

Resolving:

נניח ויש לנו תהליך או kernel שרוצה למצוא כתובת ווירטואלית. התהליך מתבצע כך:

1. תהליך או kernel מבצע Resolving של כתובת וירטואלית.
   א. MMU מקבל בקשה. (פנייה לחומרה)
   ב. אם ל-MMU יש Resolving:
      a. אחלה, קיבלנו כתובת.
   ג. אם MMU לא הצליח לעשות Resolve אבל הכתובת ממופה:
      a. נבצע Walk על הטבלה שלנו ונעדכן את TLB.
   ד. אם MMU לא הצליח לעשות Resolve וגם הכתובת אינה ממופה:
      a. נזרק PAGE_FAULT_EXCEPTION.(מתחלק לשני סוגים)
         i. SOFT- לא מצא אבל הדף קיים ושייך לתהליך אחר. אין צורך להקצות מחדש אלא למפות מחדש.(מגדילים קאונטר של הדף)
         ii. HARD- מקצים דף חדש לגמרי, ממפים ומוסיפים לטבלה.