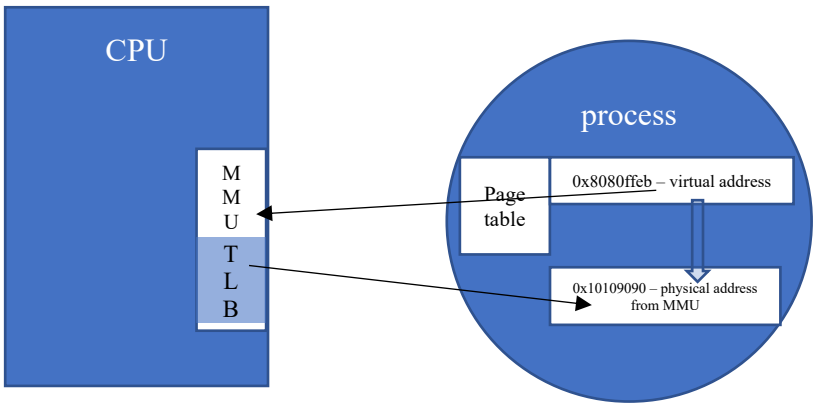
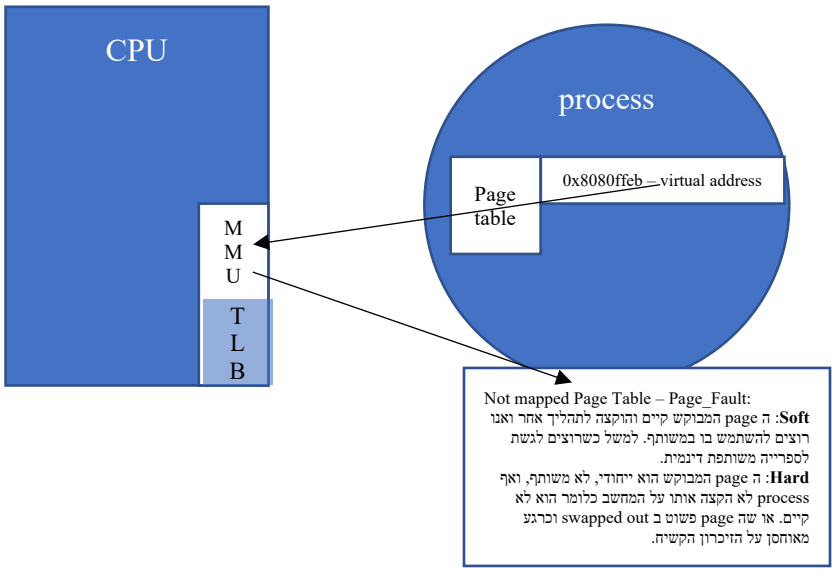
הערה חשובה משיעור קודם:

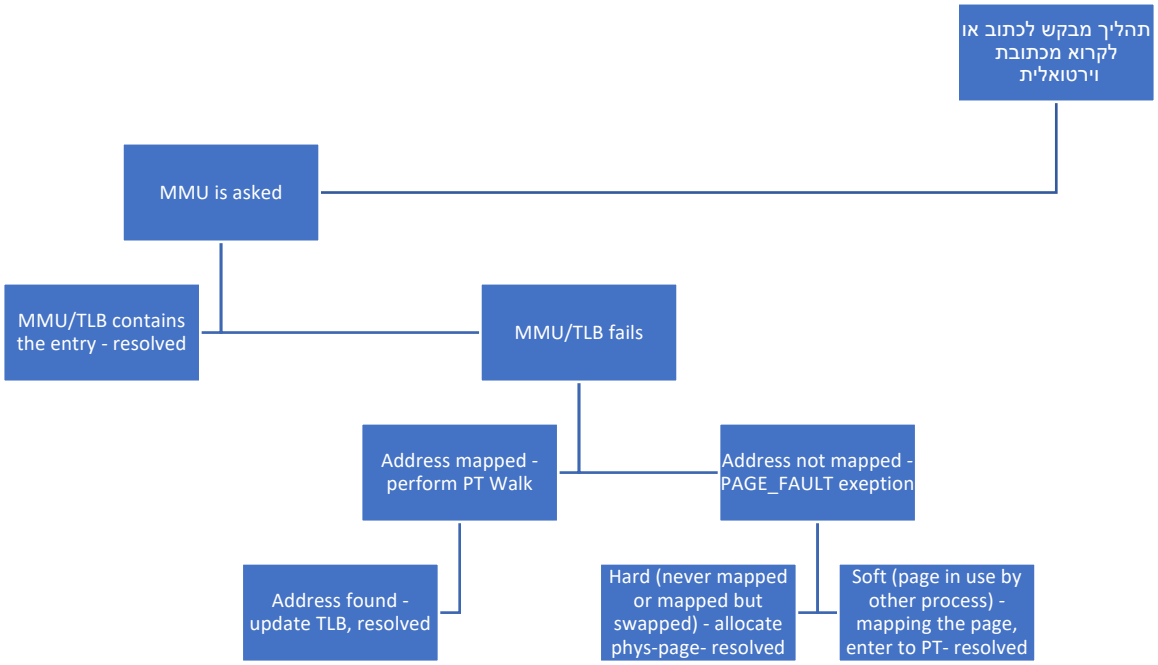
נניח שיש לנו process שמשתמש בכתובת וירטואלית. הדבר הראשון שמערכת ההפעלה עושה זה resolving מה MMU. זו הפנייה הראשונה. כעת יש שתי אפשרויות. או שהכתובת הזו הוקצתה לתהליך, וזוהי האפשרות הטובה:



האפשרות הרעה היא שה page שאליו שייכת הכתובת לא הוקצה לתהליך. (למשל אם הוא לא קיים או הוקצה לתהליך אחר. במקרה זה תזרק חריגה: Page_Fault:

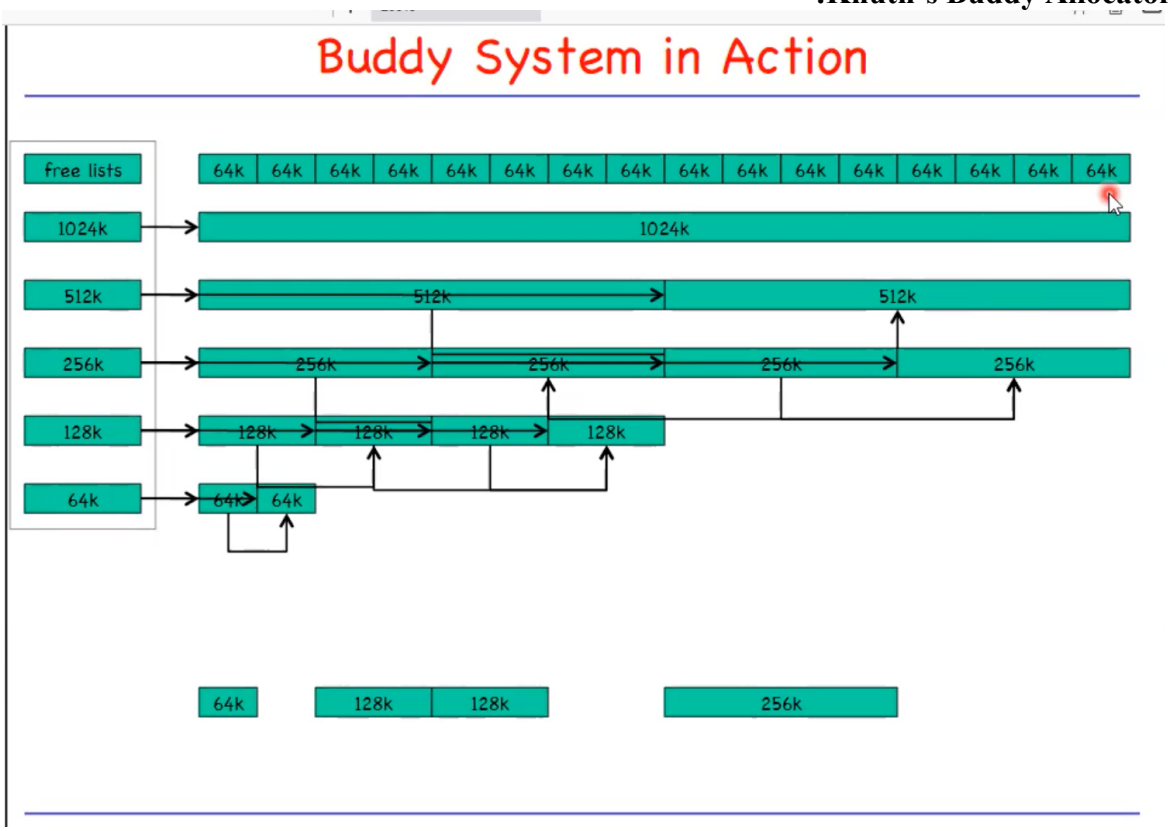


טיפול בחריגת Soft: מביאים את ה page לPage table של הprocess ומעלים באחד את ה usage counter של הpage.
טיפול בחריגת Hard: זהו המקרה שבו יש לאתחל ולמפות כתובת פיזית לכתובת וירטואלית חדשה שלא הייתה קודם בשימוש או שהייתה בשימוש ועברה לזיכרון הקשיח.
Page Table Walk: קורה כאשר ה process מבקש כתובת וירטואלית שה page שלה הוקצה אבל הכתובת לא ממופית (ב MMU אין מיפוי). לכן מחפשים ב page table כתובת. החיפוש נקרא **PT walk**. מה שקורה זה שניגשים ל PT עם הכתובת הוירטואלית ומקבלים כתובת וירטואלית חדשה שמפנה לכתובת פיזית (2 level PT אמור להיות מוסבר בשיעור קודם). לאחר התהליך הזה אפשר לשמור בTLB את המיפוי כדי לא לחזור על התהליך בפעם הבאה.
כדי לייעל ולצמצם את כמות הפעמים שצריך לעשות PT Walk, יוצרים עותק של TLB באופן לוקאלי (לאחר עדכון הTLB כמובן), ואז ה MMU מקבל resolving. הסיבה שיוצרים עותק לוקאלי ברמת ה process היא שכאשר מתבצע context switch הprocess החדש בחילוף מחליף את ה TLB שב MMU לTLB שלו וכך בעצם לא צריך למפות מחדש כתובות שהיו בשימוש.



הערה: ישנם מעבדים חדשים שיש להם MMU עם כמה כניסות בשביל כל process כלומר הם שומרים לכל pid את הTLB שלו ברמה הגלובלית וכך לא צריך להחליף את הTLB בכל context switch.
---סוף ההערה משיעור קודם---
שאלה: מה קורה כאשר קוראים ל()malloc? מתבצעת פנייה לספרייה שנקראת **GLIBC** . זו לא ספרייה בקרנל. הספרייה מחזירה זיכרון אם יש לה או מבקשת מהקרנל אם אין לה בצורה הבאה: היא קוראת לפונקציה **()mmap2** שהיא **Syscol** שמחזירה את הגודל שהוקצה ביחידות של page. לאחר מכן מתבצעת לוגיקה של **Knuth's Buddy Allocator** (מוסבר בפירוט בהמשך) כדי להקצות מתוך הדפים כתובות נצרכות. GLIBC מחזיר (כמעט) רק את כמות הכתובות שנדרשו ע"י ()malloc, ושומר את היתר עבור הקריאות הבאות.

:Knuth's Buddy Allocator



(לשם הנוחות נמחק מהטבלה את הK. כי הרעיון אותו רעיון גם עם גודל קטן יותר של page).
נניח שאנו עובדים במערכת 64bit ונדרשנו ע"י הפרוסס לאלקצ' 30 בתים של זיכרון (זה קטן מ64 ולכן נחזיר 64 שזה גודל בלוק מינימאלי), והpage שקיבלנו מה()mmap הוא בגודל 1024. מאחר ואין לנו בלוק בגודל הזה(64), (כרגע יש לנו בלוק ענק בגודל 1024) האלגוריתם יחלק ב2 את הבלוק, עד שיגיע לגודל מינימאלי עבור הבקשה.
במקרה שלנו -64. (בד"כ האלוקטור ישמור 2 בתים לפני ואחרי מרחב הכתובות שהוא עומד להקצות) **זה הבלוק אותו אנו נחזיר לפרוסס.**

סיכום ביניים: כרגע יש לנו את הבלוקים הבאים:
512: חופשי
256: חופשי
128: חופשי
64: חופשי
64: תפוס

---עד כאן מה שמתואר בתמונה, מפה זה המשך הדגמה ללא קשר לתמונה.---

לאחר מכן כאשר נקבל בקשה חדשה למשל ל70 בתים (אז צריך להחזיר 2x64), מאחר וכבר חילקנו, ויש לנו בלוק בגודל 128, נחזיר אותו.

סיכום ביניים: כרגע יש לנו את הבלוקים הבאים:
512: חופשי
256: חופשי
128: תפוס
64: חופשי
64: תפוס

נגיד שעכשיו הפרוסס שביקש את הבלוק הראשון- משחרר אותו. מה שקורה כעת זה איחוד (מרג'וג' מלשון merge). הבלוק ששוחרר מסתכל על שכנו שבגודל זהה לו. אם הוא פנוי, הם מתאחדים:

סיכום ביניים: כרגע יש לנו את הבלוקים הבאים:

512: חופשי

256: חופשי

128: תפוס

128: חופשי

נשים לב שהאיחוד קרה רק פעם אחת שכן כאשר בחנו את 128 החופשי, השכן שלו (his buddy) היה תפוס. כעת אם 128 התפוס יתבקש להשתחרר נבצע את התהליך הקודם שוב, והפעם יקרו 3 איחודים:

סיכום ביניים: כרגע יש לנו את הבלוקים הבאים:

1024: חופשי

נשים לב שעבור הקצאה האלגוריתם עובד ב$O(\log n)$ וברוב המקרים בשחרור גם כן, אבל במקרה הגרוע השחרור ידרוש איחוד כולל כמו בשחרור האחרון בדוגמא שהובאה כאן, וזמן הריצה הוא $O(n)$ כאשר n הוא גודל הpage

---עד כאן סיכום השיעור---

achiyazigi@gmail.com

יש דוגמא מצויינת בויקיפדיה שמסבירה את האלגוריתם, אמנם היא באנגלית אבל שווה להציץ (בעמוד הבא):

Step	64 K	64 K	64 K	64 K	64 K	64 K	64 K	64 K	64 K	64 K	64 K	64 K	64 K	64 K	64 K	64 K
1	2^4															
2.1	2^3								2^3							
2.2	2^2				2^2				2^3							
2.3	2^1		2^1		2^2				2^3							
2.4	2^0	2^0	2^1		2^2				2^3							
2.5	A: 2^0	2^0	2^1		2^2				2^3							
3	A: 2^0	2^0	B: 2^1		2^2				2^3							
4	A: 2^0	C: 2^0	B: 2^1		2^2				2^3							
5.1	A: 2^0	C: 2^0	B: 2^1		2^1		2^1		2^3							
5.2	A: 2^0	C: 2^0	B: 2^1		D: 2^1		2^1		2^3							
6	A: 2^0	C: 2^0	2^1		D: 2^1		2^1		2^3							

<table>
<tr><td>7.1</td><td>A: 2^0</td><td>C: 2^0</td><td>2^1</td><td>2^1</td><td>2^1</td><td>2^3</td></tr>
<tr><td>7.2</td><td>A: 2^0</td><td>C: 2^0</td><td>2^1</td><td colspan="2">2^2</td><td>2^3</td></tr>
<tr><td>8</td><td>2^0</td><td>C: 2^0</td><td>2^1</td><td colspan="2">2^2</td><td>2^3</td></tr>
<tr><td>9.1</td><td>2^0</td><td>2^0</td><td>2^1</td><td colspan="2">2^2</td><td>2^3</td></tr>
<tr><td>9.2</td><td colspan="2">2^1</td><td>2^1</td><td colspan="2">2^2</td><td>2^3</td></tr>
<tr><td>9.3</td><td colspan="3">2^2</td><td colspan="2">2^2</td><td>2^3</td></tr>
<tr><td>9.4</td><td colspan="5">2^3</td><td>2^3</td></tr>
<tr><td>9.5</td><td colspan="6">2^4</td></tr>
</table>

This allocation could have occurred in the following manner

1. The initial situation.
2. Program A requests memory 34 K, order 0.
    1. No order 0 blocks are available, so an order 4 block is split, creating two order 3 blocks.
    2. Still no order 0 blocks available, so the first order 3 block is split, creating two order 2 blocks.
    3. Still no order 0 blocks available, so the first order 2 block is split, creating two order 1 blocks.
    4. Still no order 0 blocks available, so the first order 1 block is split, creating two order 0 blocks.
    5. Now an order 0 block is available, so it is allocated to A.
3. Program B requests memory 66 K, order 1. An order 1 block is available, so it is allocated to B.
4. Program C requests memory 35 K, order 0. An order 0 block is available, so it is allocated to C.
5. Program D requests memory 67 K, order 1.
    1. No order 1 blocks are available, so an order 2 block is split, creating two order 1 blocks.
    2. Now an order 1 block is available, so it is allocated to D.

6. Program B releases its memory, freeing one order 1 block.
7. Program D releases its memory.
    1. One order 1 block is freed.
    2. Since the buddy block of the newly freed block is also free, the two are merged into one order 2 block.
8. Program A releases its memory, freeing one order 0 block.
9. Program C releases its memory.
    1. One order 0 block is freed.
    2. Since the buddy block of the newly freed block is also free, the two are merged into one order 1 block.
    3. Since the buddy block of the newly formed order 1 block is also free, the two are merged into one order 2 block.
    4. Since the buddy block of the newly formed order 2 block is also free, the two are merged into one order 3 block.
    5. Since the buddy block of the newly formed order 3 block is also free, the two are merged into one order 4 block.