מערכות הפעלה - הרצאה 11

שרון מלטר, אתגר 17

28 ביולי 2024

1 Chapter 9 Cont.

בפרקים הקודמים: למדנו על הדרך הראשונה להקצות זיכרון ראשי לתהליכים- הקצאה רציפה. נעבור לבעיית של הדרך הזאת.

1.1 בעיית הקצאה דינמית

כיצד נמלא דרישת הקצאה מרשימת חורים חופשיים? ישנן 3 שיטות לבחור חור מתאים;

- $First - fit$: נקצה את החור הראשון שגדול מספיק למלא את דרישת התהליך.
- $Best - fit$: נקצה את החור הקטן ביותר שגדול מספיק לדרישות התהליך (כמובן שבדרך זו ניצול הזיכרון הינו אופטימלי)
- $Worst - fit$: נקצה את החור הגדול ביותר שגדול מספיק לדרישות התהליך (כמובן שבדרך זו ניצול הזיכרון הינו הגרוע ביותר)

הגישות של $first - fit$ ו- $best - fit$ יותר מהירות וחסכניות מ- $worst - fit$.

וכעת לבעיה כללית יותר, פרגמנטציה.

1.2 Fragmentation

פרגמנטציה באה בשתי צורות:

- $External\ Frgmentation$ - חורים בין הקצאות זיכרון רציפות לתהליכים.
- $Fragmentation\ Internal$ - זיכרון אשר שייך לבלוק שהוקצה לתהליך אך הוא לא מנוצל על ידיו (כלומר כמות הזיכרון שהוקצתה לתהליך גדולה מהנדרש הכולל או מהנדרש ברגע מסוים)

בשיטת $First - fit$ ניתוח מראה כי כאשר מקצים N בלוקים, $0.5N$ בלוקים נאבדים לפרגמנטציה. אזי מאבדים $1/3$ מתחילת מרחב הזיכרון המיועד לתהליכים עד לבלוק האחרון שהוקצה (ישנם $1.5N$ בלוקים באזור זה)
ניתן להקטין פרגמנטציה עם $compaction$ (דחיסה), כלומר סידור כל הזיכרון החופשי בבלוק גדול אחד. דחיסה היא אפשרית רק אם ה- $relocation$ דינמי ונעשה בזמן ריצה.
כאשר יש I/O, יש להשאיר את התהליך בזיכרון ולהכניס את ה- I/O רק לבאפרים של המערכת.

עד כה דיברנו על הקצאה רציפה, אך האם היא רציפה גם בזיכרון הפיזי?

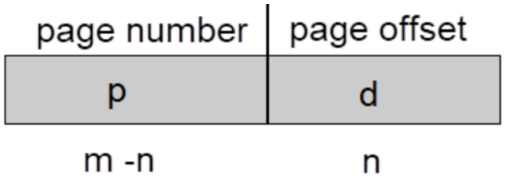
1.3 Paging

הכתובות הפיזיות של תהליך יכולות להיות לא רציפות, תהליך מקבל זיכרון פיזי לפי היכן ישנו מקום פנוי ולא בהכרח בדיוק אחרי הכתובת הקודמת שהוקצתה עבורו (אחרת תהיה בו פרגמנטציה חיצונית וגם יהיה צורך להחליט מהו טווח הכתובות הפיזיות המוקצה לתהליך)
אזי מחלקים את הזיכרון הפיזי לפריימים בגודל קבוע. הגודל הינו חזקה של 2, בין $521B$ ל- $16\ Mbytes$ וכמו כן נחלק את הזיכרון הלוגי לבלוקים בגודל זהה המכונים דפים $(pages)$, כמובן שגישה זו יכולה להיות פרגמנטציה

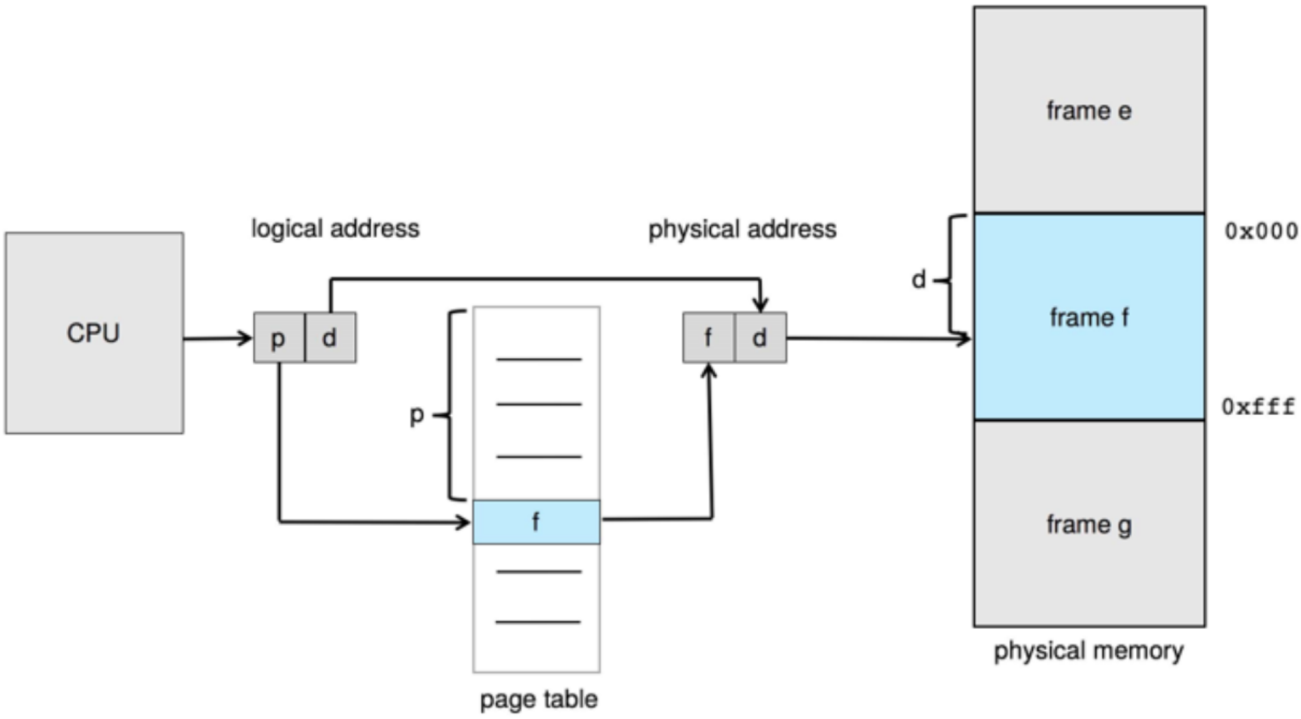
פנימית. כדי להריץ תוכנה שגודלה N דפים, נצטרך למצוא N פריימים חופשיים ולהעלות את התכנית.
כדי לתרגם כתובות לוגיות לפיזיות, משתמשים בטבלת דפים $(page\ table)$. כל כתובת שמייצר ה־ CPU (כתובת לוגית) מורכבת מהחלקים הבאים:

- $Page\ number\ (p)$ ־ מספר המשמש כאינדקס לטבלת דפים שמכילה את כתובת הבסיס של כל דף בזיכרון הפיזי. כלומר זהו המספר המציג לאיזו דף שייכת הכתובת.
- $Page\ offset\ (d)$ ־ משותף לכתובת הבסיס כדי להגדיר את הכתובת הפיזית. כלומר, זהו האינדקס של הכתובת בתוך הדף שלה.

שרטוט מורכב של כתובת לוגית:



שרטוט התהליך המורכב של יצירת ותרגום כתובת לוגית:



ננבור בנושא טבלת הדפים.

1.3.1 מימוש טבלת הדפים

כל טבלת דפים שמורה בזיכרון הראשי, כך שתהליכים יוכלו לגשת אליה מהר.

- מצביע בשם $Page-table\ base\ register\ (PTBR)$
- הערך $Page-table\ length\ register\ (PTLR)$ שומר את הגודל של טבלת הדפים (שימו ♡: כמצוין, גודל כל הדפים זהה וקבוע)

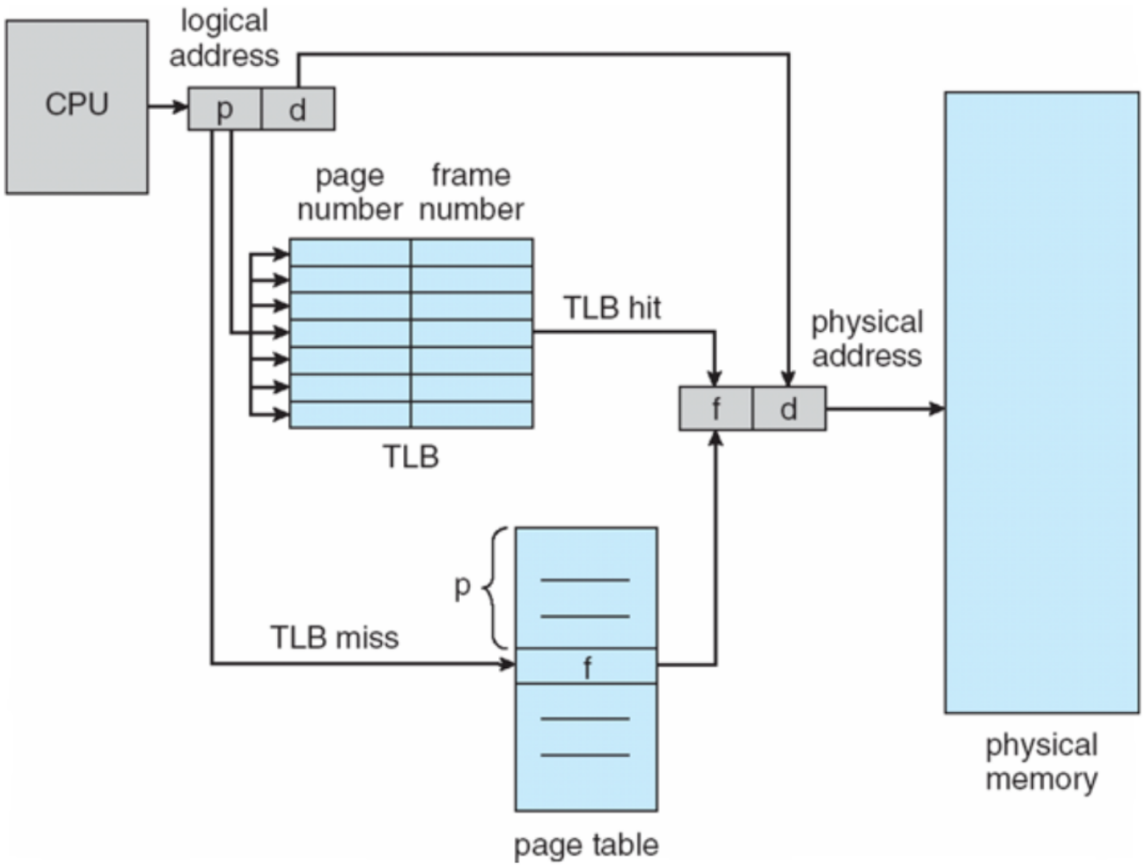
עם טבלת דפים, עבור כל גישה לפקודה או דאטה יש לבצע שתי גישות לזיכרון, אחת לטבלת הדפים ושנייה לפקודה/דאטה.
כדי לא תמיד לבצע שתי גישות, ישנם $caches$ הנקראים $translation\ look-aside\ buffers\ (TLBs)$ (מכונה גם זיכרון אסוציאטיבי)

1.4 TLB

חלק מה־ $TLBs$ שומרים $address-space\ identifiers\ (ASIDs)$ בכל כניסה, אשר מבדילים בין התהליכים כדי להגן על מרחב הכתובת שלו. אחרת היינו צריכים לנקות את כל ה־ TLB (לבצע $flush$) בכל $context\ switch$.
לרוב $TLBs$ הם קטנים וכוללים רק בין 64 ל־ 1024 כניסות. כאשר מתקיים $miss$, כלומר כאשר מחפשים כניסה שלא קיימת ב־ TLB, היא מועלית אליו על מנת לזרז את החיפוש הבא. יש כמובן להגדיר כיצד בוחרים על אלו כניסות מוותרים כאשר נגמר המקום ומנגד, יש כניסות אשר יכולות להישאר בכל מקרה על מנת לוודא חיפוש מהיר תמידי.
שרטוט מתוסבך של תכולת ה־ TLB;

Page #	Frame #

והתהליך השלם של $paging$ עם TLB;



עם TLB הגענו לנושא של זמן חיפוש. כיצד ניתן לבדוק את האפקטיביות שלו?

1.5 $Effective\ Access\ Time\ (EAT)$

$Hit\ ratio$ הוא אחוז הפעמים שבהם מספר דף נמצא ב־ TLB. באופן טבעי, $Miss = (100 - Hit)\%$ ונסמן את זמן הגישה לזיכרון ב־ T.
$effective\ access\ time$ הוא תוחלת זמן הגישה לכתובת מתהליך. כלומר:

$$EAT = Hit \cdot T + Miss \cdot 2T$$

עד כאן איכות, אבל מה לגבי החיים עצמם?

1.6 $Memory\ Protection$

הגנת הזיכרון נעשית ע"י $protection\ bit$ שמייצגת האם ניתן רק לקרוא מהפריים או גם לכתוב אליו. ניתן להוסיף ביטים נוספים כך שניתן לציין שהדף הינו להרצה בלבד וכו'.
כמו כן לכל כניסה בטבלת הדפים ישנו $Valid - invalid\ bit$. המצב "$valid$" אומר שהדף המקושר הוא במרחב הלוגי של התהליך, כלומר הוא דף חוקי (שאיננו שייך לתהליך אחר או מכיל ערכי זבל) כאשר מנסים לגשת מטבלת הדפים לדף בו $invalid$ מקבלים $page\ fault$.
המצב "$invalid$" מצביע על כך שהדף לא נמצא במרחב הכתובות הלוגי של התהליך.
במקום שני ביטים אלה ניתן להשתמש ב־ $page - table\ length\ register\ (PTLR)$, אשר מציין מהו הדף החוקי האחרון בטבלת הדפים של תהליך.

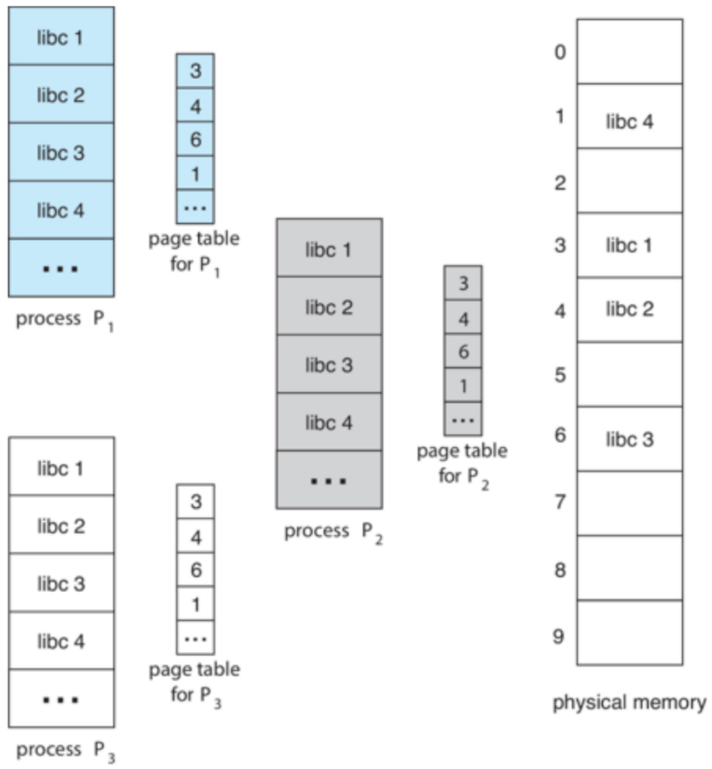
אבל כמו שלמדנו, תהליכים יכולים לחלוק ספריות. כיצד עושים זאת עם דפים?

1.7 $Shared\ Pages$

ישנם שני מקרים:

- $Shared\ code$ ־ ישנו קופי יחיד שהינו $read - only$ (כזכור נקרא גם $reentrant$) של הקוד שניתן לחלוק בין תהליכים. למשל כאשר מספר תהליכים מריצים את אותה התכנית וכמו שמספר חוטים משתמשים חולקים את אותו מרחב התהליך.
- $Private\ code\ and\ data$ ־ כל תהליך שומר קופי נפרד של הקוד והדאטה. הדפים של הקוד והדאטה יכולים להופיע בכל מקום במרחב הכתובות הלוגי.

דוגמה ויזואלית;



במקרה זה התהליכים חולקים את הקוד שבספרייה.

חמוד. אבל יש יותר מדרך אחת לבנות טבלת דפים.

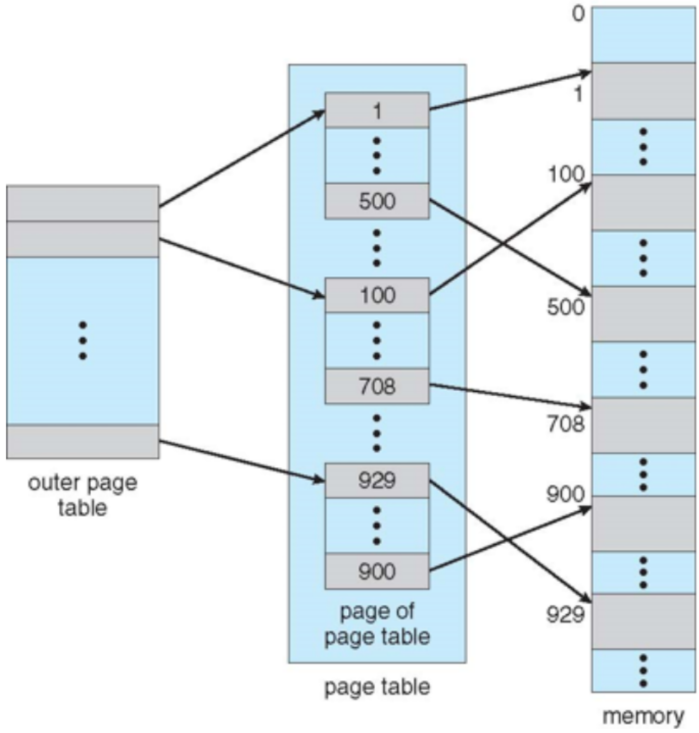
1.8 מבנה של טבלת דפים

נראה דוגמה שמציגה מתי ייתכן שנרצה מבנה פחות ישיר של טבלת דפים.
נניח שיש לנו מרחב כתובות פיזי של $32-bit$ וגודל דף של $4KB$. כל טבלת דפים תצטרך מיליון כניסות (הרי מספר הדפים הוא $\frac{2^{32}}{2^{12}} = 2^{10}$) גודל הכניסה הוא לכל הפחות $4B$ מכיוון שזהו מספר הביטים המינימלי על מנת למספר את הדפים, כך שכל תהליך יצטרך $4MB$ רק בשביל לשמור את טבלת הדפים שלו, כך שיהיה בעייתי להקצות לו זיכרון רציף.
פתרון פשוט לבעיה זו הוא לחלק את טבלת הדפים ליחידות קטנות יותר. נלמד שלוש דרכים לעשות זאת:

- $Hierarchical\ Paging$
- $Hashed\ Page\ Tables$
- $Inverted\ Page\ Tables$

1.8.1 טבלת דפים היררכית

במודל זה מפרקים את טבלת הדפים שהכרנו למספר טבלאות דפים קטנות יותר. למשל, לטבלת דפים דו־שלבית $(two-level)$
להלן דוגמה ויזואלית ודוגמה של חלוקת הכתובת לתתי הטבלאות;



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 is paged, the page number is further divided into:

- a 10-bit page number
- a 10-bit page offset

Thus, a logical address is as follows:

page number		page offset
p_1	p_2	d
10	10	12

where p_1 is an index into the outer page table, and p_2 is the displacement within the page of the inner page table

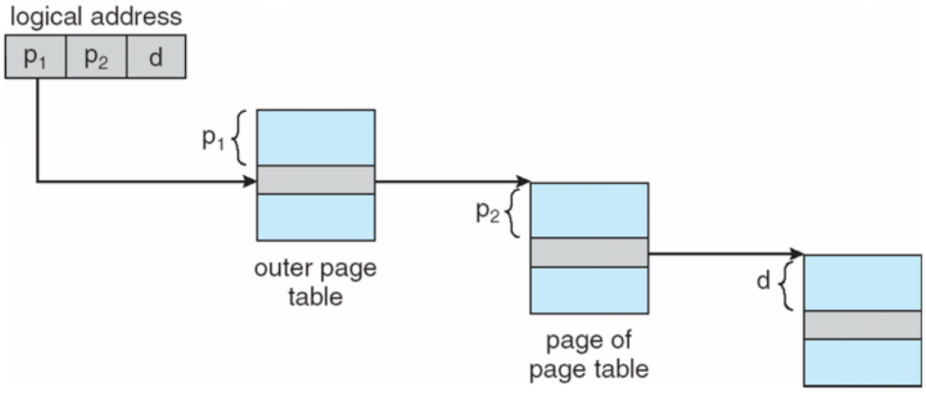
Known as **forward-mapped page table**

אכן יש שיפור, אך הוא לא משמעותי עבור גודל דף של $4KB$ ומרחב כתובות לוגי $64-bit$. מעבר לכך, באופן כללי מרחב הכתובות הלוגי הוא $32-bits <$.

1.8.2 טבלת דפים מגובבת

במבנה זה טבלת הדפים הווירטואלים מגובבת לטבלת דפים. טבלת הדפים הזו מכילה שרשרת של אלמנטים המגובבים לאותו המקום. בכל אלמנט שמור;

1. מספר הדף הווירטואלי.
2. הערך הפריים הממופה.

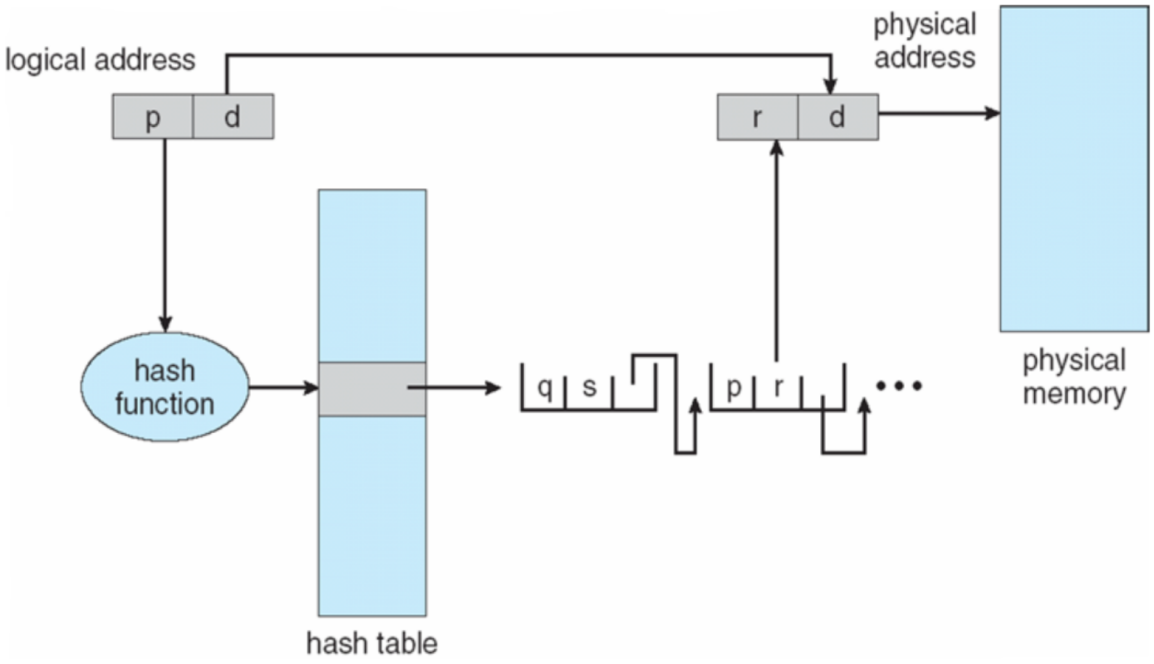


3. מצביע לאלמנט הבא.

מספרי הדף הווירטואלי מושווים בשרשרת על מנת לחפש התאמה. אם נמצאת אחת, אז הפריים הפיזי המקביל נבחר.

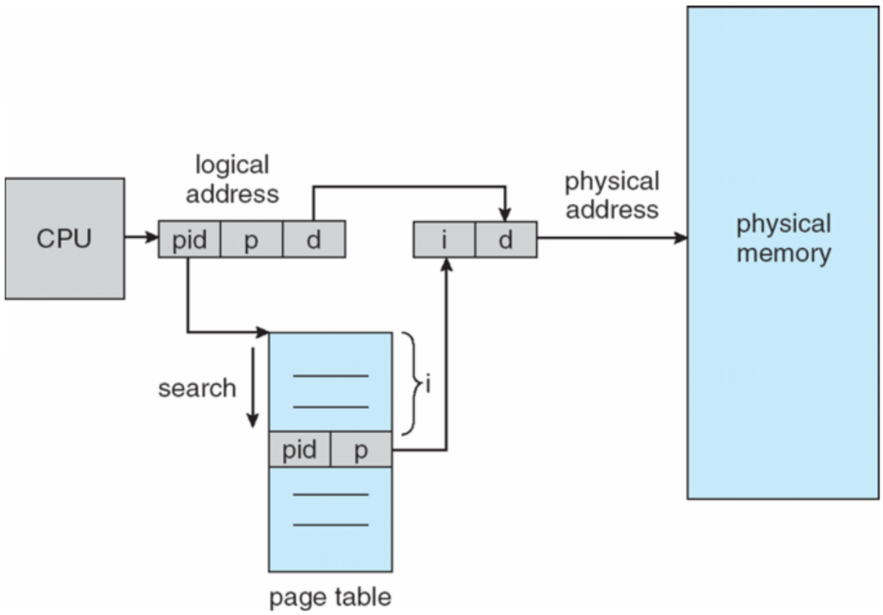
ישנה גירסא אלטרנטיבית לכתובות $64-bit$ אשר נקראת $clustered\ page\ tables$.
מדובר ברעיון דומה, אך כל כניסה מתייחסת למספר דפים במקום לאחד.
גירסא זו מתאימה גם למרחבי כתובות מפוזרים, בהם ישנה פרגמנטזציה רבה.

שרטוט של טבלת דפים מגובבת;



1.8.3 טבלת דפים הפוכה

במקום שלכל תהליך נשמור טבלת דפים ונעקוב אחרי כל הדפים האפשריים שייתכן שהוא משתמש בהם, נעקוב אחרי הדפים הפיזיים. כלומר, ישנה כניסה לכל דף אמיתי בזיכרון.
הכניסה מורכבת מהכתובת הווירטואלית של הדף ששמור באותו מקום אמיתי בזיכרון עם מידע על התהליך שמחזיק באותו דף.
מבנה זה מפחית את כמות הזיכרון הנדרשת לכל טבלת דפים, אבל מגדילה את הזמן שלוקח לעבור על על הטבלה כאשר ישנו רפרנס לדף. בכדי לקצר זמן זה, אפשר להשתמש בטבלת גיבוב אשר מגבילה את החיפוש לאחד, או מעט, כניסות. כמו כן ה־ TLB יכול לסייע.
ואיך מממשים זיכרון משותף? ־ עם מיפוי של כתובת וירטואלית לכתובת הפיזית המשותפת. להלן שרטוט הארכיטקטורה:



עד כאן מבני טבלאות דפים של תהליכים.

ולנושא הבא: מה לגבי עצירת תהליכים, כלומר *swapping*?

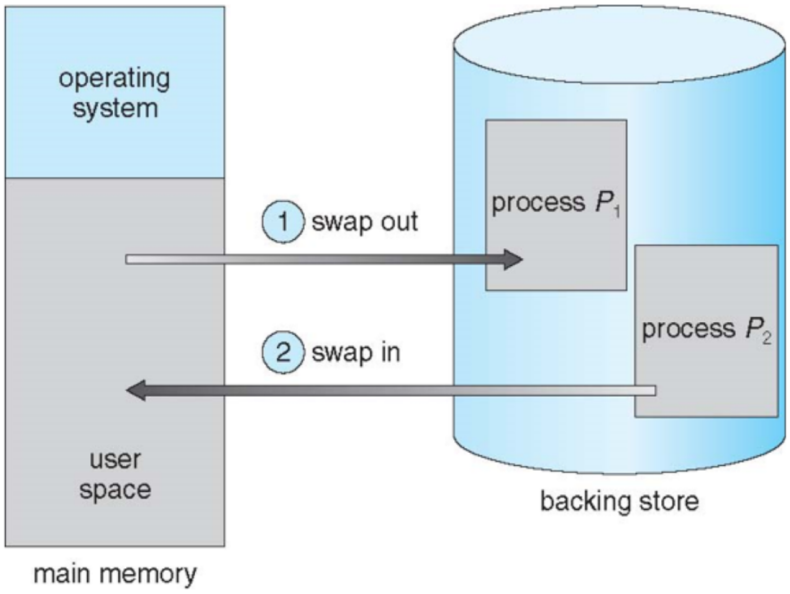
1.9 *Swapping*

ניתן להוציא תהליך מהזיכרון באופן זמני ולשמור אותו ב־ *backing store*. משם ניתן להחזיר אותו לזיכרון הראשי בשביל שימשיך לרוץ. עם זאת, כאן כמובן נוצרת בעיה בה הזיכרון הנדרש לכלל התהליכים עולה על הזיכרון החופשי.
ה־ *backing store* הוא דיסק מהיר שגדול מספיק על מנת לארח העתקים של כל תמונות הזיכרון בשביל כל היוזרים. כמו כן הוא חייב לספק גישה ישירה לכל אותן תמונות זיכרון. המערכת מנהלת *ready queue* שבו תהליכים המוכנים לרוץ אשר יש עבורם תמונות זיכרון בדיסק.
Roll out, roll in ־ זהו גירסא אחת ל־ *swapping* אשר משומשת לאלגוריתמי *scheduling* מבוססי עדיפות. באופן טבעי תהליכים בעלי עדיפות נמוכה מוחלפים החוצה כך שבעלי עדיפות גבוהה יכולים להיות מועלים ולרוץ.
זמן רב מ־ *swapping* מיועד להחלפה (*transfer time*)

להלן דוגמה לשיטת *swapping* נפוצה ומורכבת יותר;

- במצב רגיל, לא ניתן לבצע *swapping*.
- מתאפשר *swapping* אם עברנו על חסם של הקצאת זיכרון.
- המנגנון מבוטל שוב כאשר דרישת הזיכרון יורדת בחזרה מתחת לחסם.

עולה שאלה מעניינת־ האם בהכרח נרצה שלאחר *swap* נחזיר את התהליך גם עם אותן הכתובות הפיזיות?
התשובה תלויה בשיטת ה־ *address binding* (שאלה מכשילה)
להלן תרשים מתוחכם המייצג *swapping*;



כידוע, *swapping* הינו חלק מ־ *context switch*. איך הוא משפיע על הזמן הנדרש?
אם התהליכים הבאים שיש לשים ב־ *CPU* הם לא בזיכרון, יש להחליף בינם לבין תהליך אחר שכן בזיכרון. להלן דוגמה לזמן של *context switch*;

100MB process swapping to hard disk with transfer rate of 50MB/sec

- Swap out time of 2000 ms
- Plus swap in of same sized process
- Total context switch swapping component time of 4000ms (4 seconds)

ניתן להקטין זמן זה אם מקטינים את גודל הזיכרון המתחלף־ כלומר אם יודעים כמה זיכרון באמת מנוצל. ניתן ליישם זאת באמצעות קריאות המערכת $request\_memory()$ ו־ $release\_memory()$.

שימו ♡:
עד כה לא דיברנו על $swapping$ עקב I/O, מכיוון שאם זה מתבצע אז ה־ I/O מתקבל אצל תהליך לא נכון. עם זאת, כן ניתן לבצע $swapping$ אם תמיד מעבירים את ה־ I/O למרחב הקרנל ולאחר מכן להתקן I/O. טריק זה מכונה $double\ buffering$ והוא מוסיף $overhead$.

והחלק האחרון של הרצאה זו, $swapping$ עם $Paging$. $Behold$: