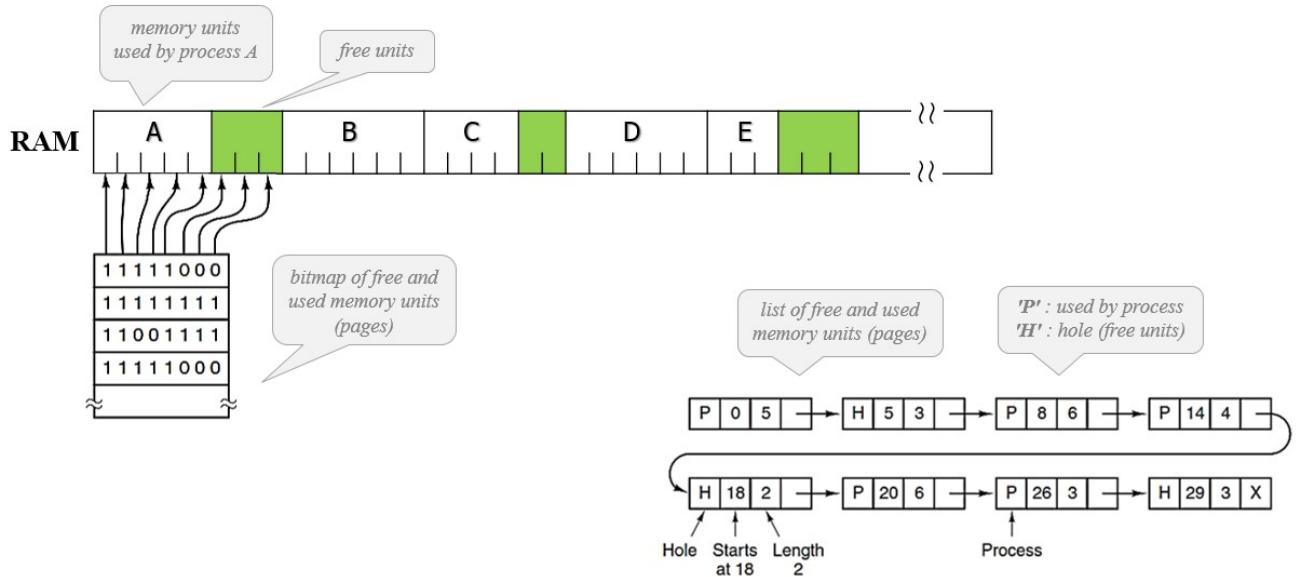


Memory Management

Tracking of allocated memory units
keeping track of used / free memory



© 2022 מרכז הפעלה

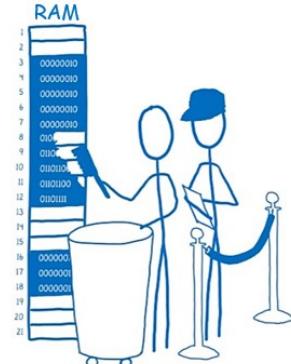
Operating Systems

Lecture 7 – Memory Management – part 1

Dr. Marina Kogan-Sadetsky

Course Syllabus

- 1. Introduction
- 2. Process Management
- 3. Scheduling algorithms
- 4. Synchronization
- 5. **Memory Management**
 - Paging
 - Multi-level paging
 - TLB
 - inverted Page Table
- 6. File Systems
- 7. Virtualization



אנו נדון בשאלת כיצד מערכת הפעלה מנהלת את הזיכרון.

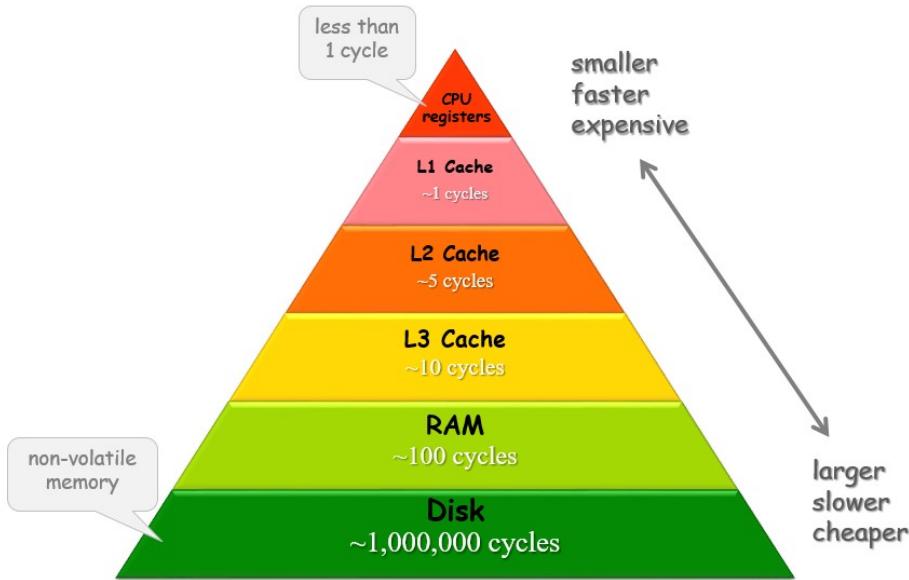
הנושאים המרכזיים:

- א. מעקב אחר הקצאות זיכרון.
- ב. מיפוי זיכרון בرمמה 1 ובמספר רמות.
- ג. TLB
- ד. טבלת דפים הפוכה
- ה. היררכיות זיכרון

Memory management: outline

- Memory – intro
- Paging
 - Multi-level paging
 - TLB & Inverted Page Table

Memory Hierarchy



הרכיב השני בחשיבותו בכל מחשב הוא הזיכרון. גם משתמשים וגם כמתכנתים, חשוב לנו כי יש "מספיק זיכרון", שהזיכרון עצמו יהיה מהיר וזול. הבעיה היא שלא קיימת בימינו טכנולוגיה העומדת על כל דרישות אלו ולכן יש צורך באגישה אחרת. למדנו ב- (MIPS) את הגישה לפיה מערכת הזיכרון היא בצורה של היררכיה כאשר המדריך המרכזי הוא מידת הקירבה הפיזית של הזיכרון לחישוב של המעבד.

הרגיסטרים הם הזיכרון המהיר ביותר, מכיוון שהם ממוקמים פיזית בתוך המעבד עצמו. המטמוןים, שהם פחות מהירים מהרגיסטרים, ממוקמים ברמות שונות. הזיכרון המרכזי מוחלק לשורות מטמון, כאשר שורות המטמון בשימוש תדיר נמצאות ברמות העליונות.

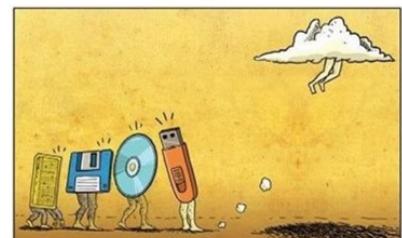
כל שהמטמון ממוקם ברמה הגבוהה יותר (כלומר, המספר שליד ה-L קטן יותר), הוא קרובה יותר למעבד ולכן יותר יתיר ביחס לرمאות התחתונות יותר...

הזיכרון הראשי, או ה-RAM, הוא משותף למספר מעבדים והוא פחות מהיר מהמטמון. לבסוף, הדיסק הוא הזיכרון האיטי ביותר. בזמן שהמעבד ממתין לקריאה מהדיסק, הוא יכול לבצע מיליון פעולות חישוב.

לכן, אחת האופטימיזציות שנעשות בניהול הזיכרון היא החלפת הקשרים. זה מאפשר לתהילך להמשיך לעבוד בזמן שננותנים מושכים מהדיסק.

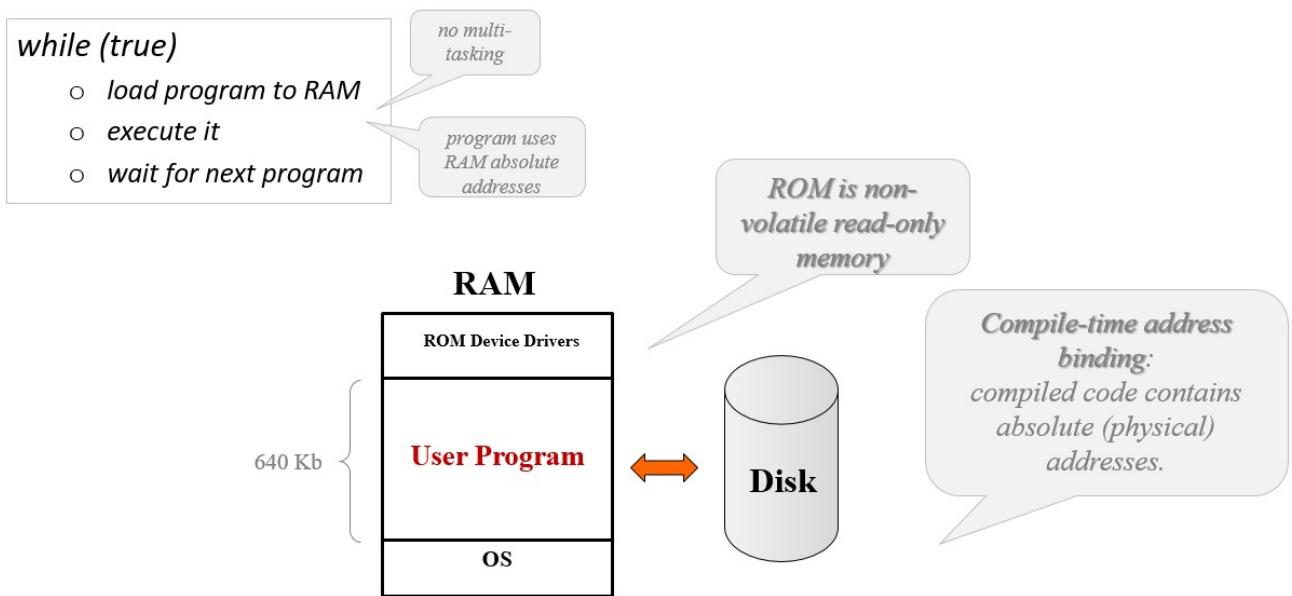
הערה: קיימים במחשב זכרונות נוספים שאינם רלוונטיים לקורס שלנו, כמו SOS Flash memory, rom, CMOS וכו'.

In order to understand memory management mechanism, we should familiarize memory management evolution



ניהול של זיכרון וירטואלי הוא דבר מאד מסובך וחזק לשימוש בפעולות מורכבות. לפני כן כדאי להתחל בהיכרות בהיסטוריה של הזיכרון.

Mono-programming: MS-DOS



בתחילת שנות ה-80, במערכת הפעלה MS-DOS, ניהול הזיכרון היה פשוט מאוד. רק תהליך אחד יכול היה להימצא בזיכרון הראשי (RAM) בכל פעם. לא הייתה אפשרות לריבוי משימות - כלומר, לא היו מספר אפליקציות שמתחרות על מקום ב-RAM.

לכן, כבר בשלב הקומpileציה, הקומpileר הגדר את הכתובת האבסולוטית שבה המעבד היה משתמש בזמן הריצה. ניהול הזיכרון היה פשוט מעלה תוכנית ל-RAM, מרים אותה, ואז ממתין לתוכנית הבאה.

הקוד המתאר את התהליך הזה הוא כדלקמן:

```

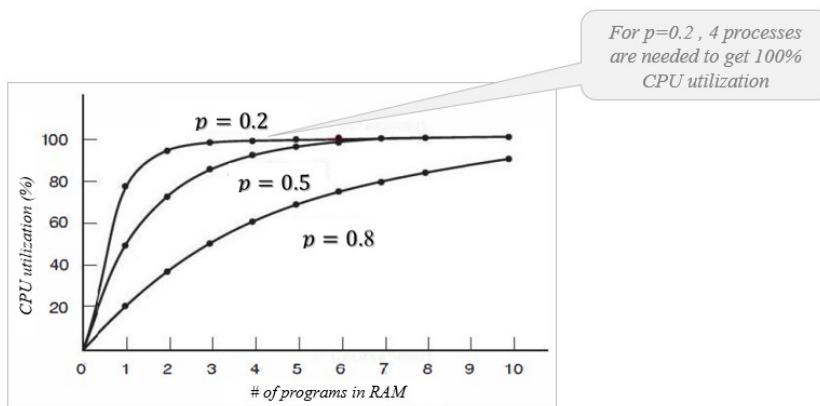
while (true)
    load program to RAM
    execute it
    wait for next program

```

במערכת זו, לא היה רבוי משימות, והתוכנית השתמשה בכתובות אבסולוטיות של ה-RAM. ה-ROM הוא זיכרון קריאה-בלבד שאינו מתמה (volatile), וה קישור של הכתובת בזמן הקומpileציה התרבצע כאשר הקוד המקומפלט הכיל כתובות פיזיות אבסולוטיות.

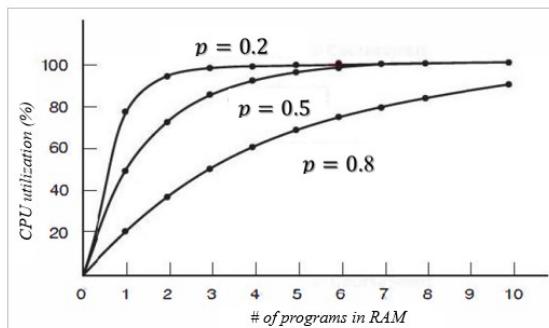
Multi-programming motivation

- p – probability for a process to wait for I/O
- p^n – probability for n processes to wait for I/O simultaneously
- → CPU utilization is $1 - p^n$



Multi-programming motivation

- p – probability for a process to wait for I/O
- p^n – probability for n processes to wait for I/O simultaneously
- → CPU utilization is $1 - p^n$



Example:

- **RAM = 1 Mb**
- each process needs 200 Kb of RAM
- OS needs 200 Kb of RAM
- $p = 0.8$

→ $1 \text{ Mb} / 200 \text{ Kb} = 5 \text{ processes, including OS}$
→ CPU utilization = $1 - 0.8^4 = 0.6 = 60\%$

- **RAM = 2 Mb**

→ $2 \text{ Mb} / 200 \text{ Kb} = 10 \text{ processes, including OS}$
→ CPU utilization = $1 - 0.8^9 = 0.87 = 87\%$

אחד הפרמטרים המשפיעים במיוחד מתקנות מרובה-תהליכיים הוא נצילות המעבד. אם יש לנו תהליך אחד שמתוין להתקן קלט-פלט, המעבד נאלץ להמתין עד שהtagובה מההתקן מגיעה.

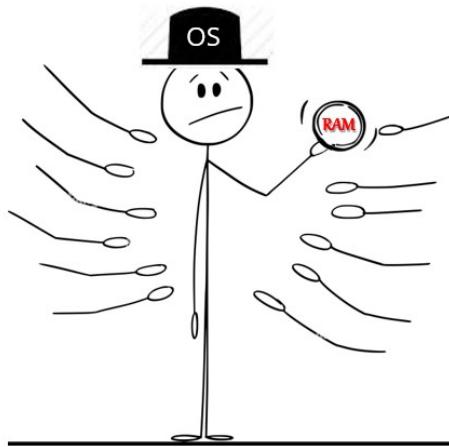
אם P הוא ההסתברות שתתהליך ימתין לפעולות קלט-פלט (IO), אז נצילות המעבד היא ההסתברות של לפחות אחד לא ימתין לפעולות IO. לדוגמה, אם $P=0.2$, אז ארבעה תהליכיים נדרשים כדי להגיעה לנצילות של 100% של המעבד.

נניח שאודל ה-RAM הוא 1 מגהבייט, כל תהליך דרוש 200 קילובייט, ומערכות הפעלה דורשת 200 קילובייט. כאמור, בכל רגע נתון יכולים להיות ב-RAM חמישה תהליכיים (כולל מערכת הפעלה). אם ההסתברות שתתהליך ימתין לפעולות IO היא 0.8, אז נצילות המעבד היא 0.6, שהיא 60%.

אם ה-RAM היה יכול להיות גדול יותר, אז היו יכולים להיות יותר תהליכיים ב-RAM. זה משפר את נצילות המעבד. במילים אחרות, זיכרון RAM גדול יותר

Multiple processes need to cooperatively use RAM.

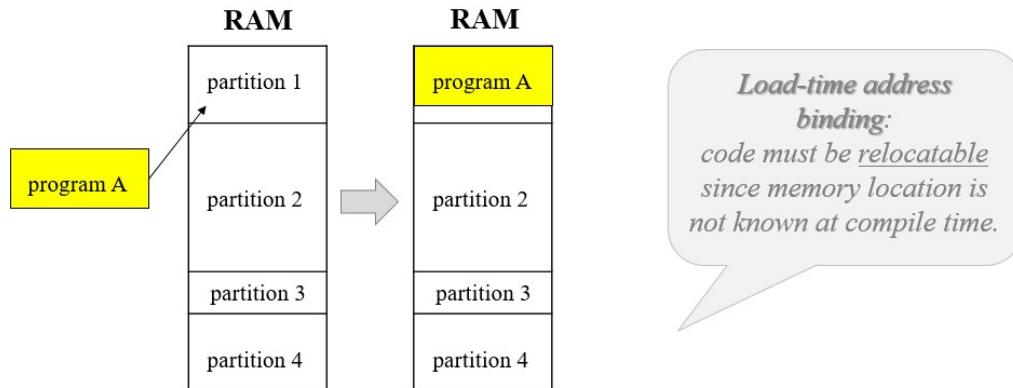
How to divide RAM between them ?



כיצד ניתן לחלקם בראם?
כיצד ניתן לחלקם בראם?

Multi-processing with Fixed RAM partitions

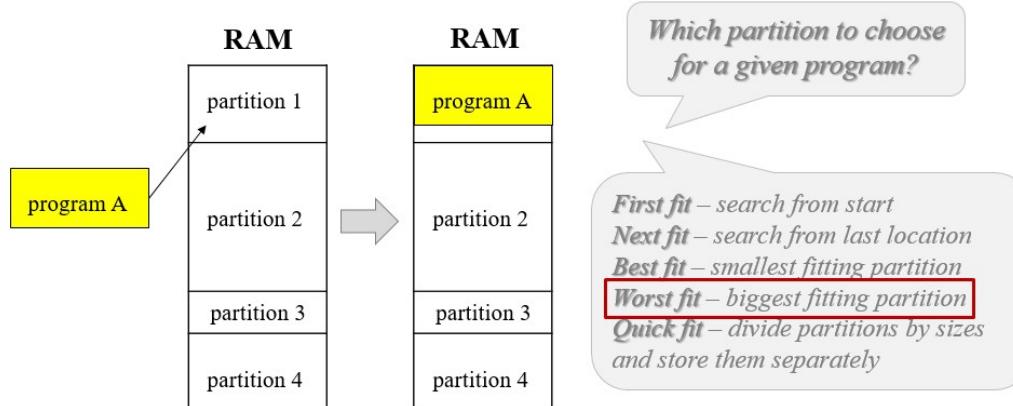
- divide RAM into fixed-size partitions
- process may be loaded into partition iff $|process| \leq |partition|$



*Load-time address binding:
code must be relocatable
since memory location is
not known at compile time.*

Multi-processing with Fixed RAM partitions

- divide RAM into fixed-size partitions
- process may be loaded into partition iff $|process| \leq |partition|$

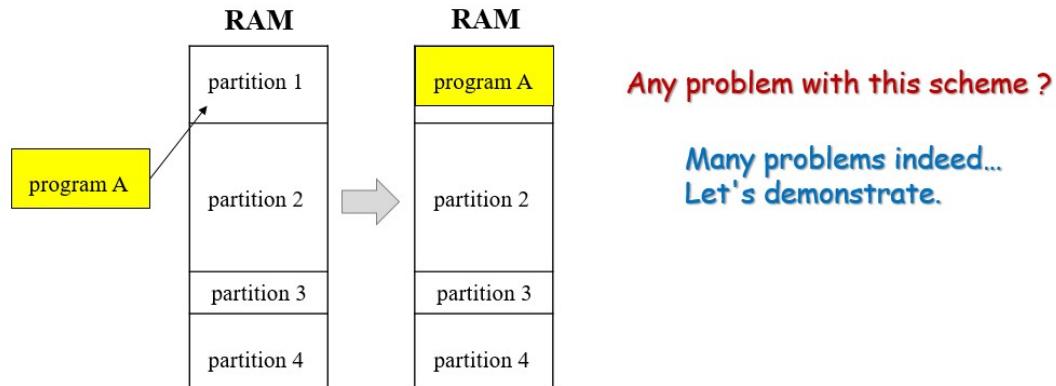


Why worst fit ??

Because the space left in partition after process loading may be enough to load some other process.

Multi-processing with Fixed RAM partitions

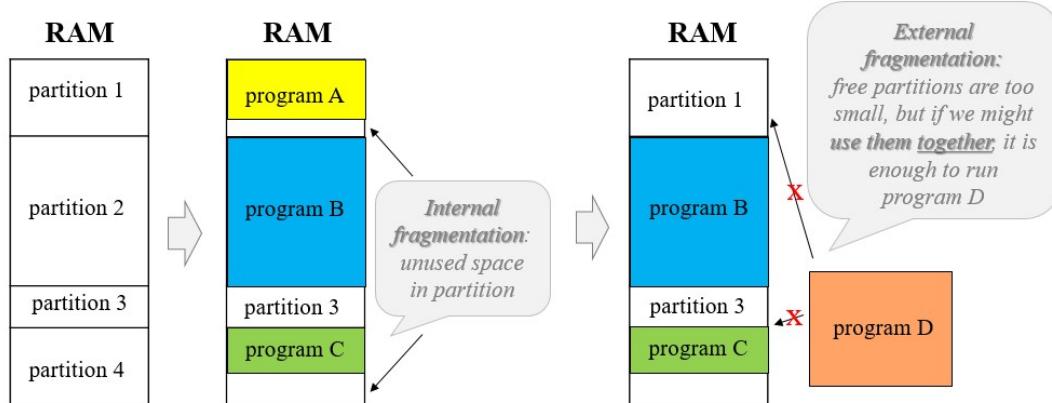
- divide RAM into fixed-size partitions
- process may be loaded into partition iff $|process| \leq |partition|$



Many problems indeed...
Let's demonstrate.

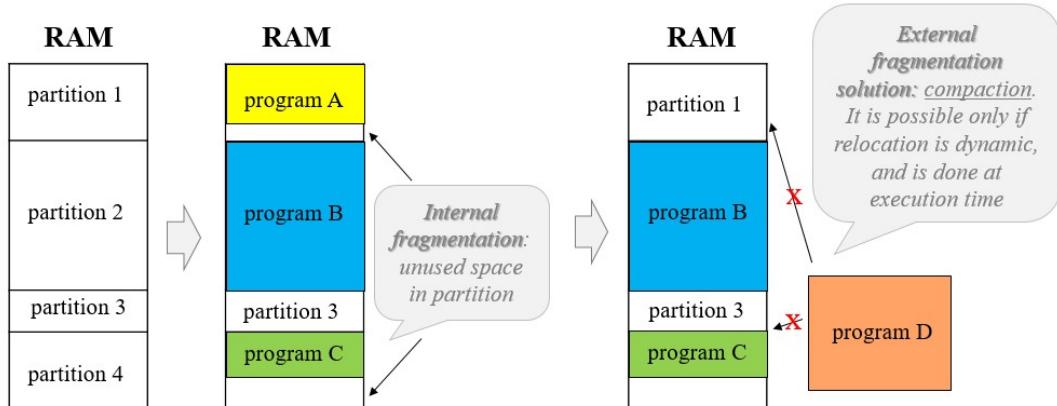
Multi-processing with Fixed RAM partitions

- divide RAM into fixed-size partitions
- process may be loaded into partition iff $|process| \leq |partition|$



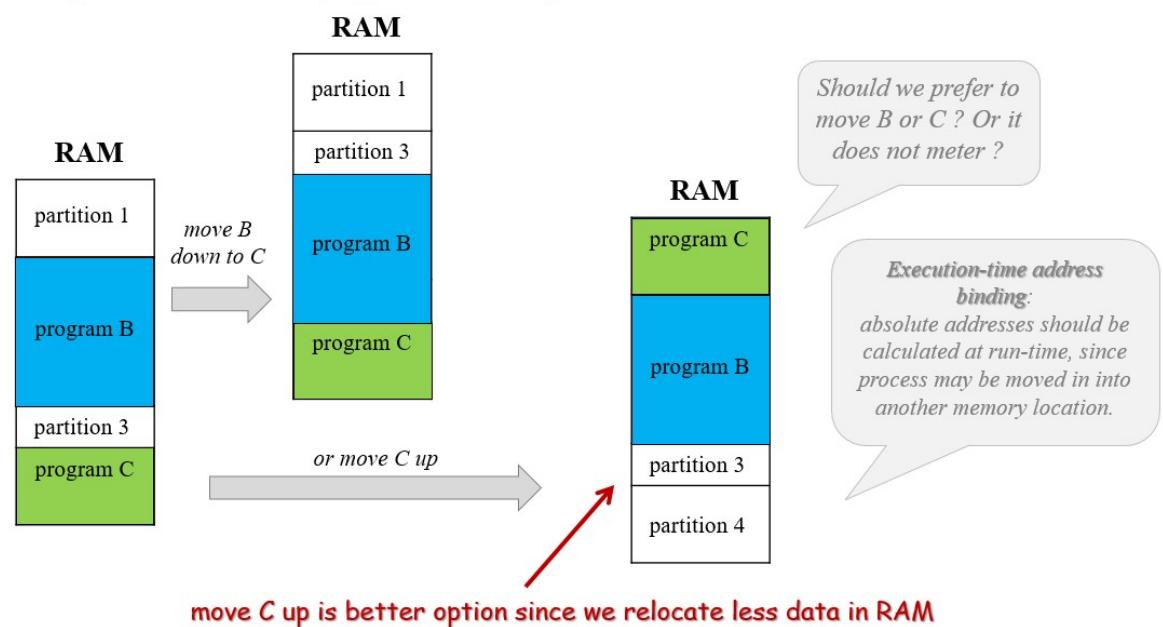
Multi-processing with Fixed RAM partitions

- divide RAM into fixed-size partitions
- process may be loaded into partition iff $|process| \leq |partition|$



External fragmentation solution: compaction

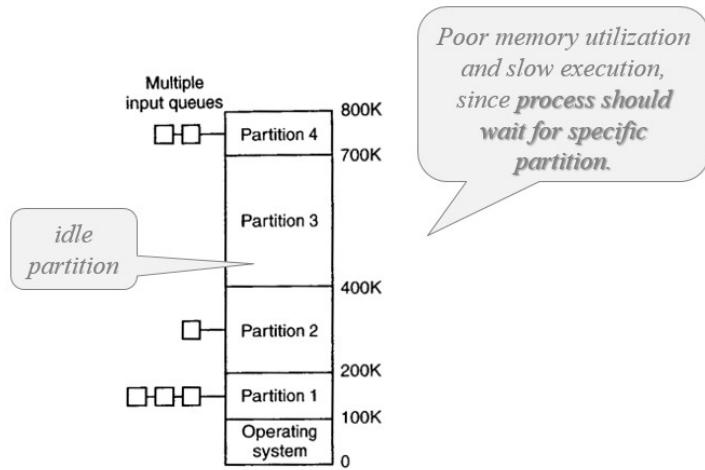
- gather all free memory together in one large block



Fixed partitions usage example:

IBM OS/360 (MFT)

- fixed RAM partitions
- queue of processes per partition
- insert process P into smallest partition that fits P



גישה 1: חלוקת הזיכרון למחיצות בגודל קבוע

בגישה זו כאשר משתמש מבקש להריץ תהליך מסוים, המערכת בוחרת מחיצה שמתאימה לאודל של התהילר. אפשר לבדוק את זה דרך הפורמט של קובץ ההרצאה שכן זה משווה שהקומפайлר יודע לחשב. אם יש מחיצה פנינה שמתאימה, התהילר יטען ל-RAM.

בשלב זה, הקומפайлר לא יכול כבר להחילט על הכתובות הפיזיות של ה-RAM, ולכן התפקיד מועבר ל-loader, שתשלים את הכתובות.

איך נבחר את המחיצה המתאימה?

איןטואיטיבית - נשיר אראן פנוי קטגוריה עם הכí הרבה פריטים.

ישנן מספר אפשרויות:

- א. התאמה ראשונה - חיפוש מההתחלתה.
- ב. התאמה הבאה - חיפוש מהמיקום האחרון.
- ג. התאמה הטובה ביותר - המחיצה הכí קטנה שמתאימה.
- ד. התאמה הגורעה ביותר - המחיצה הכí גדולה שמתאימה.
- ה. התאמה מהירה - חלוקת המחיצות לפי אדלים ושמירה עליהם בנפה.

למה הכלל של התאמה הגורעה ביותר עובדת?

כי המקום שנשאר במחיצה לאחר טיעינת התהילר עשוי להיות מספיק לטיעינת התהילר אחר.

מי שראה אי פעם מחשן בצבא יכול לתרاء לעצמו עד כמה החלוקת הזאת אינה הגיונית - יכול להיות שתהיה לי קטgorיה שלמה של פריטים שלא תוכל להיכנס לארגזים כיון ואנו מאפשרים לכל אראן להחזיק קטגוריה אחת בלבד. מה יקרה אם צה"ל יחליט להשתמש בקטgorיה של אבקת חמל?

הבעיה בשיטת החלוקת זו היא שם יש מקום פנוי ב-RAM, הוא יכול להיות מפוזר בין תוכניות. כתוצאה לכך, יתכן מצב שבו תהליך שcn יש לו מספיק מקום ב-RAM, לא יוכל להיכנס כי אין מקום פנוי רציף. לשם כך אנו מאחדים את ה-RAM כך שהזיכרון הפנוי יהיה רציף.

הפתרון לפיצול החיצוני הוא דחיסה. זה אפשרי רק אם ההזזה היא דינמית, ומתבצעת בזמן הריצה. ניתן להזיז את B למיטה ל-C או להזיז את C למעלה. הזרת C לעליה היא האפשרות הטובה יותר מאחר שאנו מזיזים פחות נתונים ב-RAM. האם עדיף להזיז את B או C? או שזה לא משנה?

במהלך זמן הריצה, כתובות מוחלטות צריכות להיחס בזמן הריצה, לאחר והתהילר יכול להיז במקומות זיכרון אחר.

The Buddy Algorithm 1963 by [Harry Markowitz](#) low external fragmentation and fast compaction

- choose best fit partition and halve it recursively until get smallest fit
- recursively merge freed partition with its neighbors of same size



Harry Max Markowitz

The Buddy Algorithm 1963 by Harry Markowitz

ב-1963, הארי מרקובייץ' פרסם אלגוריתם לניהול זיכרון שמשתמש בשני הגישות לחלוקת ה-RAM, עם הוכחה מלאה. האלגוריתם מחלק את הזיכרון לבlocks בגודלים שהם חזוקות של 2 ומשתמש במערכת מיוחדת למקבץ אחר כל בלוק. הוא בוחר בכל שלב את המחיצה המתאימה ביותר ומהיצה אותה באופן רקורסיבי עד שמאע למחיצה בגודל המתאים. כאשר מחיצה משוחררת, האלגוריתם ממזג מחיצות פנויות באופן רקורסיבי עם שכינהיהם באותו הגודל.

במילים אחרות - יצירת המחיצות מתבצעת באופן דינמי.

יתרונות:

- הказאות הזיכרון הן ייעילות מאוד כשהאודל שנשאר הוא בהכרח קטן מהאודל של התהילר. עם מימוש נכון, תהליך הקצאת הזיכרון יכול להיות מהיר.
- מספר החלוקות קטן.
- קל למימוש.

חסרונות:

- יש פיצולים פנימיים.
 - על הזיכרון הדרש לאלגוריתם - הבזבוז כאן הוא פחות או יותר כמו האקלום יש overhead.
 - הגדלים של הבלוקים מוגבלים לחזקות של 2. מה אם הקטגוריה החדשה שאנו חונכו מנסים להכניס למיחסן?
- האלגוריתם אינו בשימוש, אך לינוקס משתמש ברעיון של המערכת שמנחת את האלגוריתם לאלגוריתמים שלו למיפוי זיכרון.

The Buddy Algorithm

1963 by [Harry Markowitz](#)
low external fragmentation and fast compaction

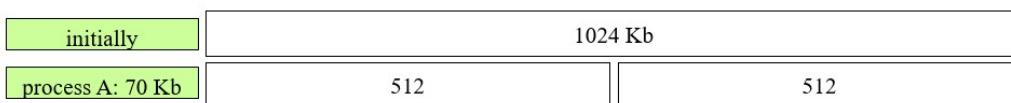
- choose best fit partition and halve it recursively until get smallest fit
- recursively merge freed partition with its neighbors of same size



The Buddy Algorithm

1963 by [Harry Markowitz](#)
low external fragmentation and fast compaction

- choose best fit partition and halve it recursively until get smallest fit
- recursively merge freed partition with its neighbors of same size

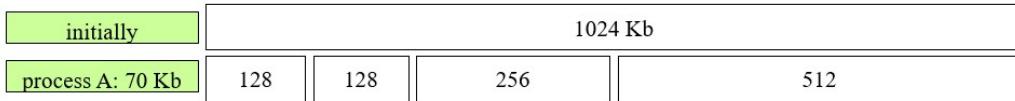


The Buddy Algorithm

1963 by [Harry Markowitz](#)

low external fragmentation and fast compaction

- choose best fit partition and halve it recursively until get smallest fit
- recursively merge freed partition with its neighbors of same size

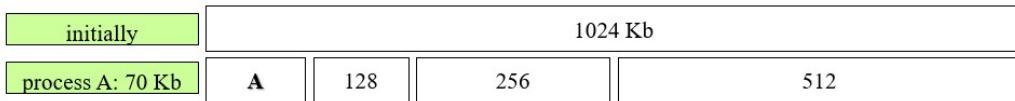


The Buddy Algorithm

1963 by [Harry Markowitz](#)

low external fragmentation and fast compaction

- choose best fit partition and halve it recursively until get smallest fit
- recursively merge freed partition with its neighbors of same size



The Buddy Algorithm

1963 by [Harry Markowitz](#)

low external fragmentation and fast compaction

- choose best fit partition and halve it recursively until get smallest fit
- recursively merge freed partition with its neighbors of same size

initially	1024 Kb			
process A: 70 Kb	A	128	256	512
process B: 35 Kb	A	64	64	256

The Buddy Algorithm

1963 by [Harry Markowitz](#)

low external fragmentation and fast compaction

- choose best fit partition and halve it recursively until get smallest fit
- recursively merge freed partition with its neighbors of same size

initially	1024 Kb			
process A: 70 Kb	A	128	256	512
process B: 35 Kb	A	B	64	256

The Buddy Algorithm

1963 by [Harry Markowitz](#)

low external fragmentation and fast compaction

- choose best fit partition and halve it recursively until get smallest fit
- recursively merge freed partition with its neighbors of same size

initially	1024 Kb				
process A: 70 Kb	A	128	256	512	
process B: 35 Kb	A	B	64	256	512
process C: 80 Kb	A	B	64	128	128

The Buddy Algorithm

1963 by [Harry Markowitz](#)

low external fragmentation and fast compaction

- choose best fit partition and halve it recursively until get smallest fit
- recursively merge freed partition with its neighbors of same size

initially	1024 Kb				
process A: 70 Kb	A	128	256	512	
process B: 35 Kb	A	B	64	256	512
process C: 80 Kb	A	B	64	C	128

The Buddy Algorithm

1963 by [Harry Markowitz](#)

low external fragmentation and fast compaction

- choose best fit partition and halve it recursively until get smallest fit
- recursively merge freed partition with its neighbors of same size

initially	1024 Kb				
process A: 70 Kb	A	128	256	512	
process B: 35 Kb	A	B	64	256	512
process C: 80 Kb	A	B	64	C	128
A is terminated	128	B	64	C	128
					512

The Buddy Algorithm

1963 by [Harry Markowitz](#)

low external fragmentation and fast compaction

- choose best fit partition and halve it recursively until get smallest fit
- recursively merge freed partition with its neighbors of same size

initially	1024 Kb				
process A: 70 Kb	A	128	256	512	
process B: 35 Kb	A	B	64	256	512
process C: 80 Kb	A	B	64	C	128
A is terminated	128	B	64	C	128
process D: 60 Kb	128	B	D	C	128
					512

The Buddy Algorithm

1963 by [Harry Markowitz](#)

low external fragmentation and fast compaction

- choose best fit partition and halve it recursively until get smallest fit
- recursively merge freed partition with its neighbors of same size

initially	1024 Kb				
process A: 70 Kb	A	128	256	512	
process B: 35 Kb	A	B	64	256	512
process C: 80 Kb	A	B	64	C	128
A is terminated	128	B	64	C	128
process D: 60 Kb	128	B	D	C	128
B is terminated	128	64	D	C	128

The Buddy Algorithm

1963 by [Harry Markowitz](#)

low external fragmentation and fast compaction

- choose best fit partition and halve it recursively until get smallest fit
- recursively merge freed partition with its neighbors of same size

initially	1024 Kb				
process A: 70 Kb	A	128	256	512	
process B: 35 Kb	A	B	64	256	512
process C: 80 Kb	A	B	64	C	128
A is terminated	128	B	64	C	128
process D: 60 Kb	128	B	D	C	128
B is terminated	128	64	D	C	128
D is terminated	128	64	64	C	128

The Buddy Algorithm

1963 by [Harry Markowitz](#)

low external fragmentation and fast compaction

- choose best fit partition and halve it recursively until get smallest fit
- recursively merge freed partition with its neighbors of same size

initially	1024 Kb				
process A: 70 Kb	A	128	256	512	
process B: 35 Kb	A	B	64	256	512
process C: 80 Kb	A	B	64	C	128
A is terminated	128	B	64	C	128
process D: 60 Kb	128	B	D	C	128
B is terminated	128	64	D	C	128
D is terminated	128	128		C	128
					512

The Buddy Algorithm

1963 by [Harry Markowitz](#)

low external fragmentation and fast compaction

- choose best fit partition and halve it recursively until get smallest fit
- recursively merge freed partition with its neighbors of same size

initially	1024 Kb				
process A: 70 Kb	A	128	256	512	
process B: 35 Kb	A	B	64	256	512
process C: 80 Kb	A	B	64	C	128
A is terminated	128	B	64	C	128
process D: 60 Kb	128	B	D	C	128
B is terminated	128	64	D	C	128
D is terminated	256		C	128	512

The Buddy Algorithm

1963 by [Harry Markowitz](#)

low external fragmentation and fast compaction

- choose best fit partition and halve it recursively until get smallest fit
- recursively merge freed partition with its neighbors of same size

initially	1024 Kb			
process A: 70 Kb	A	128	256	512
process B: 35 Kb	A	B	64	256 512
process C: 80 Kb	A	B	64	C 128 512
A is terminated	128	B	64	C 128 512
process D: 60 Kb	128	B	D	C 128 512
B is terminated	128	64	D	C 128 512
D is terminated	256		C 128 512	
C is terminated	256		128 512	

The Buddy Algorithm

1963 by [Harry Markowitz](#)

low external fragmentation and fast compaction

- choose best fit partition and halve it recursively until get smallest fit
- recursively merge freed partition with its neighbors of same size

initially	1024 Kb			
process A: 70 Kb	A	128	256	512
process B: 35 Kb	A	B	64	256 512
process C: 80 Kb	A	B	64	C 128 512
A is terminated	128	B	64	C 128 512
process D: 60 Kb	128	B	D	C 128 512
B is terminated	128	64	D	C 128 512
D is terminated	256		C 128 512	
C is terminated	256		256	512

The Buddy Algorithm

1963 by [Harry Markowitz](#)

low external fragmentation and fast compaction

- choose best fit partition and halve it recursively until get smallest fit
- recursively merge freed partition with its neighbors of same size

initially	1024 Kb				
process A: 70 Kb	A	128	256	512	
process B: 35 Kb	A	B	64	256	512
process C: 80 Kb	A	B	64	C	128
A is terminated	128	B	64	C	128
process D: 60 Kb	128	B	D	C	128
B is terminated	128	64	D	C	128
D is terminated	256		C	128	512
C is terminated	512			512	

The Buddy Algorithm

1963 by [Harry Markowitz](#)

low external fragmentation and fast compaction

- choose best fit partition and halve it recursively until get smallest fit
- recursively merge freed partition with its neighbors of same size

initially	1024 Kb				
process A: 70 Kb	A	128	256	512	
process B: 35 Kb	A	B	64	256	512
process C: 80 Kb	A	B	64	C	128
A is terminated	128	B	64	C	128
process D: 60 Kb	128	B	D	C	128
B is terminated	128	64	D	C	128
D is terminated	256		C	128	512
C is terminated	512			512	

דוגמת ריצה:

נתון:

נתון מחשב בעל זיכרון באודל 1024 בתים.

כמו כן ישנו 3 תהליכיים כך שתהליך A הוא באודל 70 קילובטים, תהליך B באודל 35 קילובטים, תהליך C באודל 80 קילובטים ותהליך D באודל 60 קילובטים

כמו כן נתון כי הסדר פעולות הכנסה הוצאה לארם הוא ABCADBDC

כאשר תהליך A נכנס לזכרון, האלגוריתם חוצה את הראם ל-2 מחיצות בגודל 128, ממחיצה בגודל 256 ואז ממחיצה בגודל 512. מאחר ולא ניתן לחוץ עוד את הממחיצה בגודל 128, תהליך A הלהה למעשה תופס את הממחיצה הזאת.

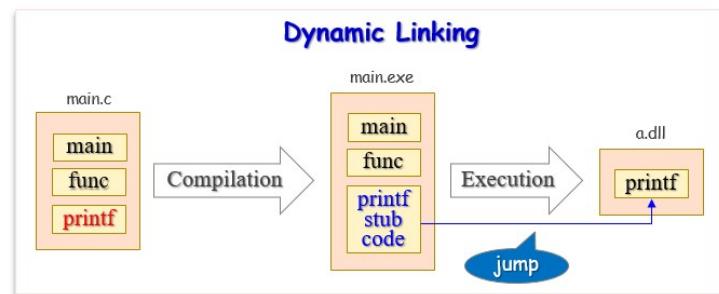
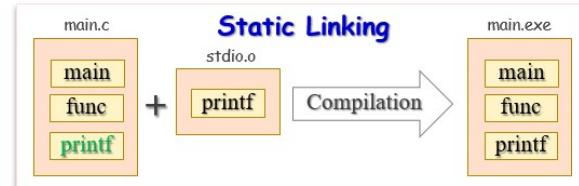
כאשר תהליך B מגיע, האלגוריתם הולך למחיצה בגודל 128 אשר פנוייה, חוצה אותה ומכו尼斯 את תהליך B לאחת המחיצות הפנויות. כאשר תהליך C מגיע, האלגוריתם חוצה את המחיצה בגודל 256 ושומר את התהליך באחת המחיצות הפנויות. כאשר מפנים את תהליך A מהזיכרון, אין עוד מחיצות פנויות מגודל 128 ולכן אין מיזוג של מחיצות סמוכות.

Dynamic Linking Libraries (DLLs)

DLLs save space in both main memory and disk

DLL allows keep only one copy of code in RAM, but this code could be used by many programs.

- ❑ DLL linking is executed at run time
- ❑ A small piece of code, **stub**, is used to locate the appropriate memory-resident DLL routine
- ❑ Stub replaces itself with the address of the routine, and calls the routine
- ❑ OS ensures the routine is mapped to process memory address



ספריות משותפות (Dynamic Link Libraries (DLL בווינדוס))

במערכות הפעלה מודרניות, כמו ווינדוס, קיימת אופטימיזציה לצמצום נפח הזיכרון בעזרת ספריות משותפות, המכונות בוינדוס (DLL). המטרה העיקרית שלהן היא לצמצם את נפח הזיכרון הדרוש להרצת התהליכים במערכת.

הספריות המשותפות משמשות את כמה תהליכים בו זמןית, ובמיוחד ספריות הקשורות לסביבת הגרפית של המערכת. אין צורך להכיל את קוד הספריות בכל תוכנה ותוכנה (כפי שבגבורינג (Static Binding)). במקום זאת, בזמן ה-linkage בקומpileציה, מוכנס stub שמנפה לספריות הרלוונטיות. בזמן הריצה, מערכת הפעלה טוענת את הקוד החדש מהספריות.

בשימוש בספריות משותפות ישנו שני יתרונות משמעותיים:

חסוך בזכרון: אם לדוגמה יש 100 תהליכים משתמשים באותה ספריה, יהיה תטען פעם אחת בלבד, וזה יחסור בזכרון.

עדכונים ותיקונים: כאשר נמצא באג או בעיה בספריה, אפשר לבדוקה פעם אחת בלבד וכל התוכנות. המשמשות בה יקבלו את הגרסה המעודכנת - בלי הצורך לкопיאן אותן מחדש.

בסיום, חשוב לציין שהשימוש בספריות המשותפות הוא בטוח, לאחר שהן בזכרון לקריאה בלבד, והן מוגנות מפני דרישת שינוי על ידי תהליכים אחרים.

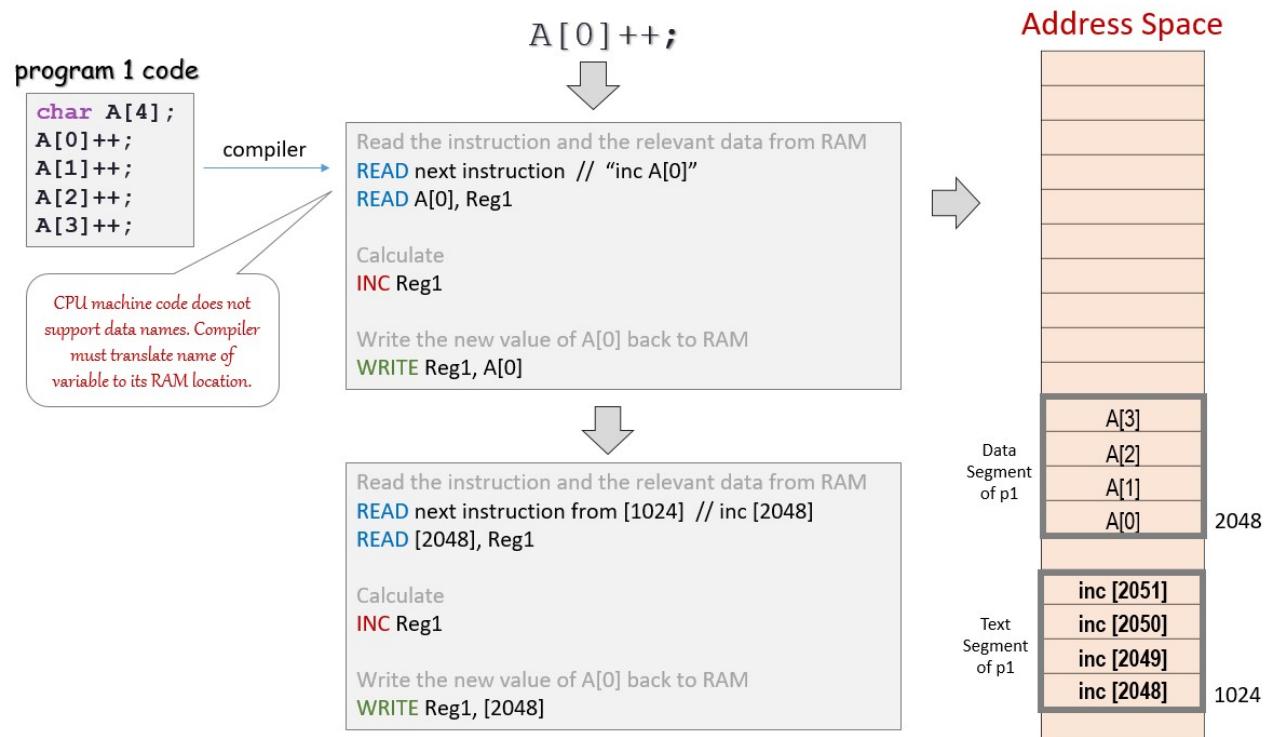
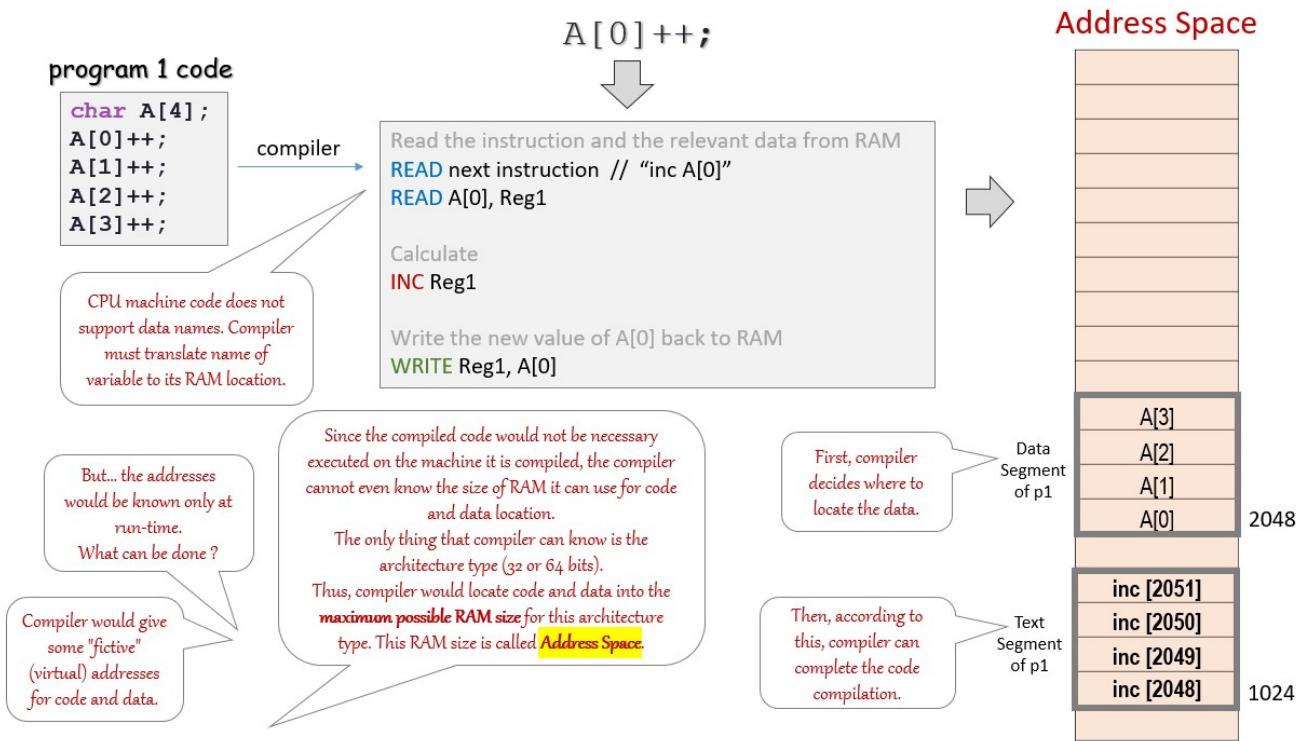
So far, we have assumed a process is smaller than memory.

What can we do if
a process is
larger than RAM ??

different memory
management
system is needed

Memory management: outline

- Memory – intro
- Paging
 - Multi-level paging
 - TLB & Inverted Page Table



Paging

עד כה, הנחנו שהתהליכים היו קטנים יותר מהזיכרון. כתוב, אנו רוצים לאפשר לתהליכיים שהם גדולים יותר מהזיכרון להתבצע במחשב. לשם כך, אנו זקוקים לסטרטגיה חדשה לניהול זיכרון. אנו נלמד על מיפוי זיכרון לדפים.

נסתכל על דוגמה שמחישה את הקשר בין קוד של תוכנית, ה-RAM והתקפיך של הקומpileר בתרגום המשתנים לכתובות שלם ב-RAM.

זכור שלכל פקודה יש חמישה שלבים: .fetch, decode, read, execute, write

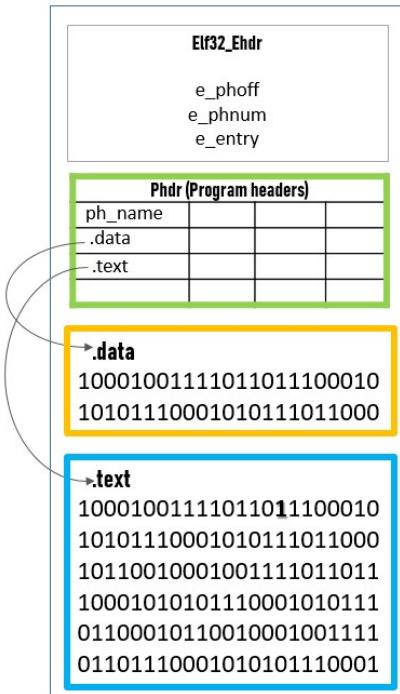
הבעיה: הכתובות הפיזיות של המנתנים ידועות רק בזמן הריצה. לשם כך, הקומפайлר מציב כתובות פיקטיביות עבור הקוד והנתנים. הכתובות הללו הן **כתובות יירטואליות**. היתרון בשימוש בכתובות יירטואליות הוא שהן מאפשרות לקוד שעבר קומפילציה להתבצע במחשבים שונים מבלי להתחשב באורך של ה-RAM. במקום זאת, הקומפайлר קובע בזמן הקומפילציה את האורך המקורי של ה-RAM לפי ארכיטקטורת המעבד, או במילים אחרות - **מרחב הכתובות**.

נניח שאנו בארכיטקטורה של 32 ביט. בארכיטקטורה זו, מרחב הכתובות הוא 4GB, אך יתכן שפיזית יש במחשב אורך שונה.

הreasון הוא שיש מרחב כתובות יירטואלי, שאין קיים במציאות. הקומפайлר ממלא כתובות בצורה מאוד קומפקטיבית: למטה את אלו שאינם משתנים ולמעלה את אלו המשתנים - stack, heap. מי שמקצה את האורך של ה-stack הוא ה-loader. כל פעם שהזיכרונו מתמלא, מתקבלת חריגה שבתוכה ממנה מערכת הפעלה יכולה להחליט האם להגדיל או להקטין את הזיכרונו.

מרחב הזיכרונו אינו קיים באמת, הוא דמיוני - הקומפайлר מוסיף כתובות לפי החלטה דמיונית זו. למיטה, ניתן לראות איך הקומפайлר מכך את הקוד ומשבץ אותו בקוד.

ELF executable file



```
marina@marina-virtual-machine:~/Desktop/Archi184/lecture3$ readelf -h a.exe
ELF Header:
  Magic: 7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  Class: ELF32
  Data: 2's complement, little endian
  Version: 1 (current)
  OS/ABI: UNIX - System V
  ABI Version: 0
  Type: EXEC (Executable file)
  Machine: Intel 80386
  Version: 0x1
  Entry point address: 0x8049000 →
  Start of program headers: 52 (bytes into file)
  Start of section headers: 8552 (bytes into file)
  Flags: 0x0
  Size of this header: 52 (bytes)
  Size of program headers: 32 (bytes)
  Number of program headers: 3
  Size of section headers: 40 (bytes)
  Number of section headers: 6
  Section header string table index: 5
```

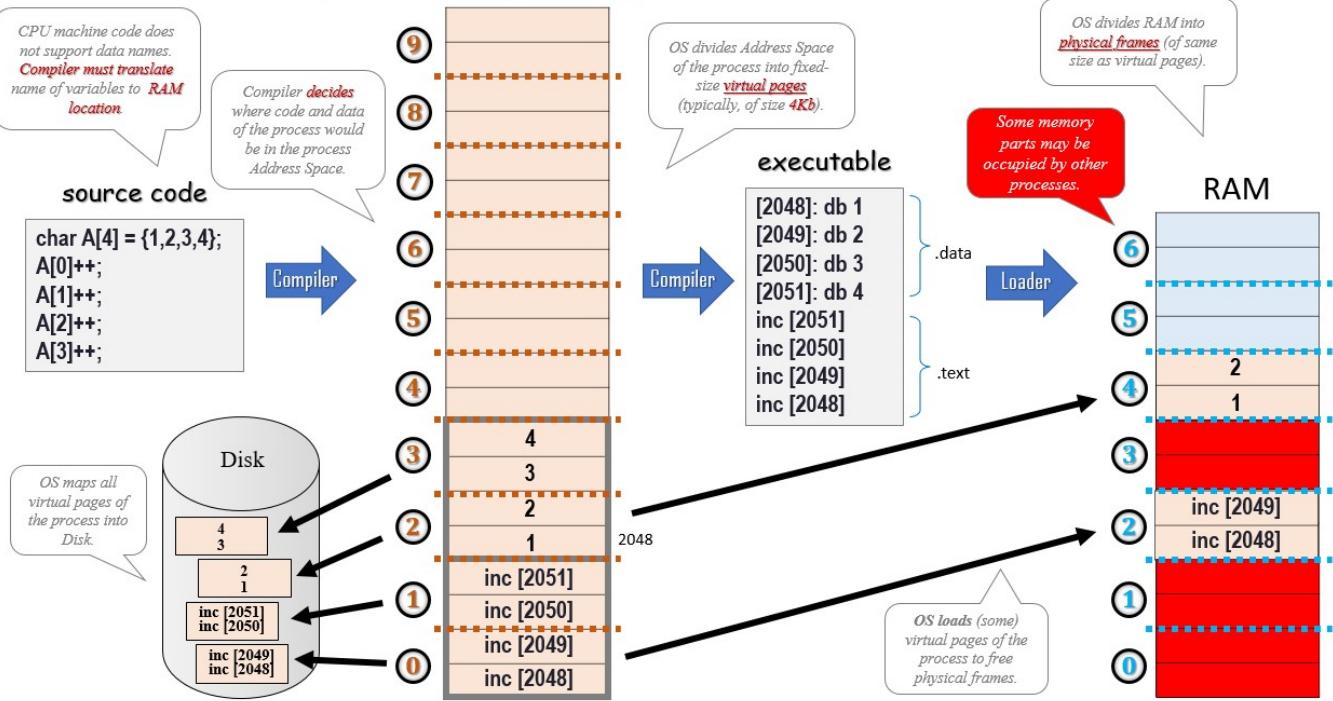
```
marina@marina-virtual-machine:~/Desktop/Archi184/lecture3$ readelf -l a.exe
Elf file type is EXEC (Executable file)
Entry point 0x8049000
There are 3 program headers, starting at offset 52

Program Headers:
  Type          Offset    VirtAddr   PhysAddr  FileSiz MemSiz Flg Align
  LOAD          0x000000 0x08048000 0x08048000 0x000094 0x000094 R  0x1000
  LOAD          0x001000 0x08049000 0x08049000 0x000049 0x000049 R E  0x1000
  LOAD          0x002000 0x0804a000 0x0804a000 0x000014 0x000014 R  0x1000

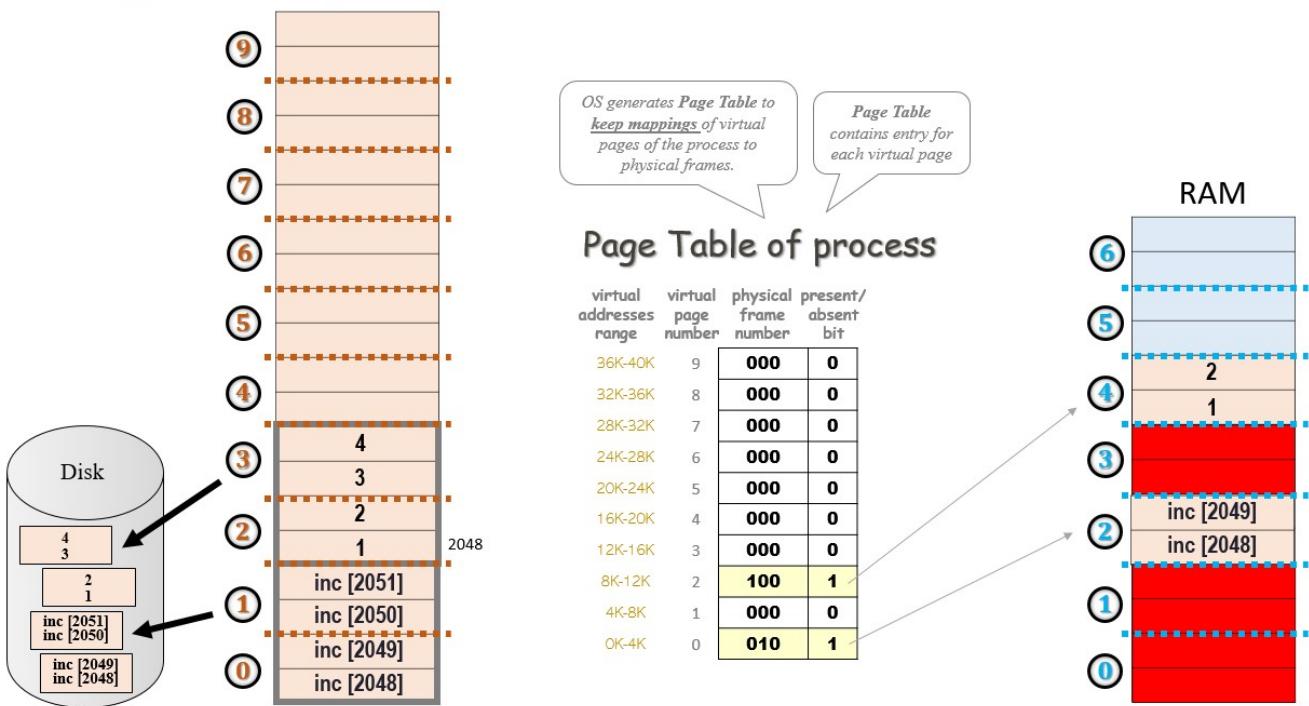
Section to Segment mapping:
  Segment Sections...
    00
    01  .text
    02  .rodata
```

E means executable

Example: Address Space of process = 40Kb



Example: Address Space of process = 40Kb



המעבד שלנו לא מכיר שמות של משתנים, אלא רק כתובות זיכרון. המשמעות של זה היא שה-Compiler צריך לתרגם את שמות המשתנים למיקומים ב-RAM. ה-Compiler מחליף איפה הקוד והנתונים של התהיליך יהיו במרחב הכתובות שלו ואז מערכת הפעלה מחלקת את מרחב הכתובות לדפים וירטואליים בגודל קבוע, מחלקת את ה-RAM לمسגרות פיזיות באותו הגודל כמו הדפים הווירטואליים, ממפה את כל הדפים הווירטואליים של התהיליך לדיסק ואז טוענת חלק מהדפים הווירטואליים של התהיליך (או כולם) למסגרות פיזיות פנויות. חשוב להבהיר כי חלקים מהזיכרון עשויים להיות תפוסים על ידי תהליכים אחרים.

נניח, לדוגמה, שמרחב הכתובות הוא בגודל של 40 קילובייט. במקרה שלנו, כאשר ה-loader רוצה להעלות את התוכנית, יתכן מצב שחלקים מהזיכרון כבר תפוסים על ידי תוכניות אחרות. מאחר ויש לנו זיכרון בגודל של 40 קילובייט וגודל כל דף הוא 4KB, ניתן בכל רגע נתון להחזיק לכל היוטר 10 דפים. אבל בדוגמה שלנו, כמות המסגרות קטנה מכמות הדפים. לכן כאשר מערכת הפעלה ממפה את הדפים לזכרון, היא גם ממפה אותם

לדיסק באזור מיוחד אשר נקרא אזור ה-swapping, כלומר כל הדפים המאוכלים נשמרים גם בדיסק באזור זה. מערכת ההפעלה מחליטה איזה דפים מתוך תמונה הזיכרון להעתה. במקרה שלנו, מערכת ההפעלה העלתה את כל הנתונים ורק חלק מההווראות (כי הם קטנים).

אז כמה מסגרות צריך בשבייל תחילת ריצה של תוכנית? אם לא נביא דפים בכלל, אז כאשר המעבד יפנה ל-RAM, הוא יקבל פסקה על כך שהתחילה לא נמצא ב-RAM ואז נקלט default.page. מערכת ההפעלה יכולה לתפוס את זה, להבין שהוא באשמה וזה תביא אותו. כמובן, אפשר בלי דפים בכלל. אבל ההיגיון הבריא אומר שאנו צריכים דף לפחות, דף למשתנים גלובליים, דף לסטטיק ודף לזיכרון הרימה (פחות 6 דפים).

Page tables

מערכת ההפעלה מחזיקה בטבלה מיוחדת, ה-table.page, כדי לנווה את הדפים. כל שורה בטבלה מייצגת דף וירטואלי אחד, כך שמערכת ההפעלה שומרת לכל דף את הנתונים הבאים: תחום כתובות וירטואליות, מספר הדף, מספר המסגרת הפיזית, בית המציג האם הדף בזיכרון או לא בזיכרון. בצורה זו מערכת ההפעלה יכולה לנווה בצורה יעילה את המיפוי של כל דפים וירטואליים למסגרות פיזיות.

נניח כי תחילת ריצה לאשת ל-1024. לשם כך מערכת ההפעלה מבצעת את הפעולה הבאה, בעזרתה היא יודעת לאיזה רשותה בטבלה לאשת:

(1024-1096)%10

בעזרת הטבלה המעבד יודע איפה ב-RAM זה נמצא, יותר מזה, המעבד חייב להחזיק מצביע עלייה כדי לאשת אליה. רוב המיקומות יהיו אפסים כי הם לא מצויים. מערכת ההפעלה מייצרת Page Table כדי לשמור על מיפויים של דפים וירטואליים של התחליר למסגרות פיזיות. ה-table.page מכילה רשומה עבור כל דף וירטואלי.

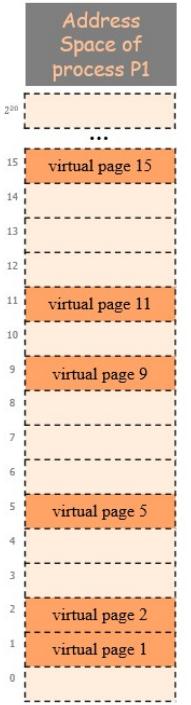
הרוב המוחלט של מערכות הזיכרון הווירטואלי משתמשות בטכנית שנ刻画ת paging. בכל מחשב, התוכניות מפנות לסת של כתובות זיכרון. כאשר תוכנית מבצעת הוראה כמו

MOV REG, 1000

היא עשויה זאת כדי להעתיק את התוכן של כתובות הזיכרון 1000 ל-REG (בהנחה שהאופרנד הראשון מייצג את היעד והשני את המקור). ניתן ליצור כתובות באמצעות אינדקסים, רישומי בסיס, ודריכים שונות אחרות. הכתובות שהתוכנית מייצרת נקבעות כתובות וירטואליות והן מהוות את מרחב הכתובות הווירטואלי. במחשבים ללא זיכרון וירטואלי, הכתובת הווירטואלית מועברת ישירות לבאס של הזיכרון וgormת למילה הפיזית של הזיכרון עם אותה הכתובת להירה או להיכתב. כאשר משתמשים בזיכרון וירטואלי, הכתובות הווירטואליות לא הולכות ישירות לבאס של הזיכרון. במקום זאת, הן הולכות ל-UML (יחידת ניהול זיכרון) שmaps את הכתובות הווירטואליות לכתובות זיכרון פיזיות.

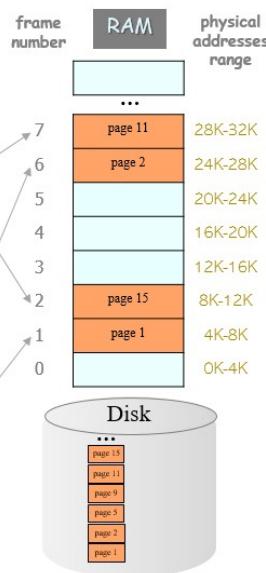
במחשבים מודרניים משתמשים בכתובות באורך של 32 או 64 ביט, השיטה שהסבירה לעליה יכולה להיות מוחלטת באותו אופן. ה-table.page תצטרך 1,048,576 (220) רשומות. במחשב עם זיכרון RAM בנפח של מספר ג'אגה-בתים, זה אפשרי. אך כתובות באורך של 64 ביט ודפים בגודל של 4KB ידרשו כ- 4.5×1015 (252) רשומות ב-table.page, מה שב哈哈לט לא אפשרי, ולכן נדרשות טכניקות אחרות.

Address Space of process $2^{32} = 4Gb$



Page Table of P1

virtual addresses range	virtual page number	physical frame number	present/absent bit
3.9G-4G	15	000	0
60K-64K	15	010	1
56K-60K	14	000	0
52K-56K	13	000	0
48K-52K	12	000	0
44K-48K	11	111	1
40K-44K	10	000	0
36K-40K	9	000	0
32K-36K	8	000	0
28K-32K	7	000	0
24K-28K	6	000	0
20K-24K	5	000	0
16K-20K	4	000	0
12K-16K	3	000	0
8K-12K	2	110	1
4K-8K	1	001	1
0K-4K	0	000	0



In 64-bit architecture,
Address Space of
process is $2^{32} = 4Gb$

Thus, Address Space
is divided into
 $4Gb / 4Kb = 2^{20}$
virtual pages

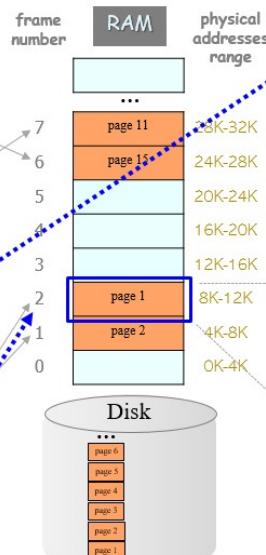
This means that Page
Table of process has
 2^{20} entries

Let's demonstrate the
Page Table usage
during process
execution.



Page Table of P1

virtual addresses range	virtual page number	physical frame number	present/absent bit
3.9G-4G	15	000	0
60K-64K	15	110	1
56K-60K	14	000	0
52K-56K	13	000	0
48K-52K	12	000	0
44K-48K	11	111	1
40K-44K	10	000	0
36K-40K	9	000	0
32K-36K	8	000	0
28K-32K	7	000	0
24K-28K	6	000	0
20K-24K	5	000	0
16K-20K	4	000	0
12K-16K	3	000	0
8K-12K	2	001	1
4K-8K	1	010	1
0K-4K	0	000	0



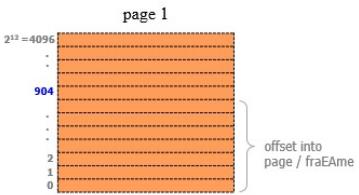
5000 is virtual address

$4Kb = 4096 \text{ bytes} < 5000 < 8192 \text{ bytes} = 8Kb$
→ virtual address 5000 belongs to virtual page #1

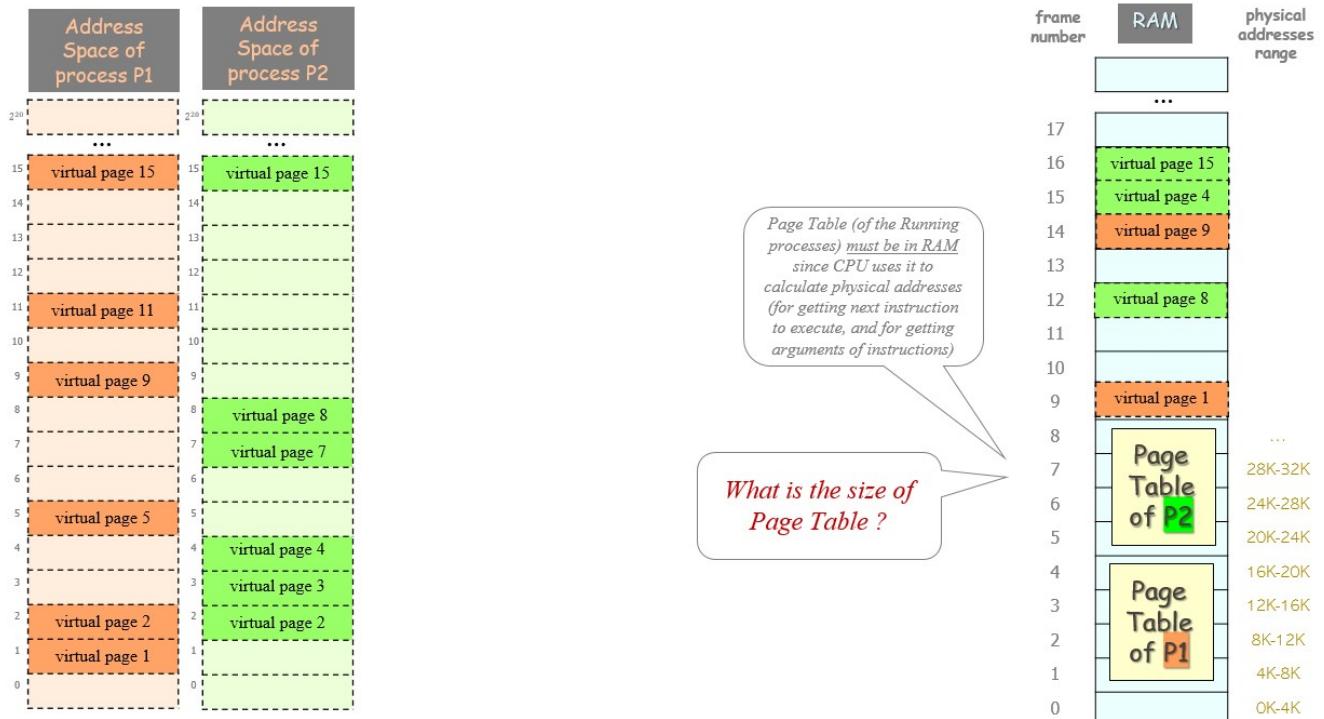
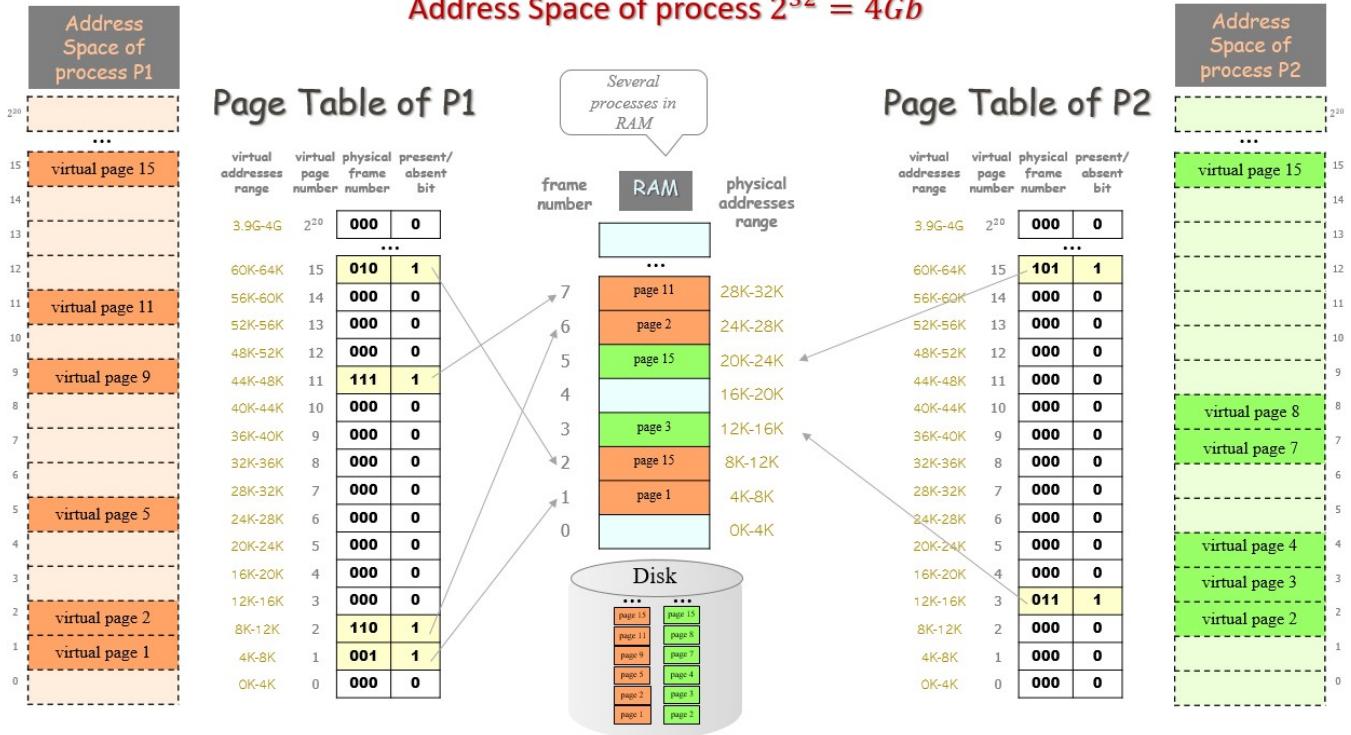
→ go to the record #1 of the Page Table
virtual page #1 is in frame #2

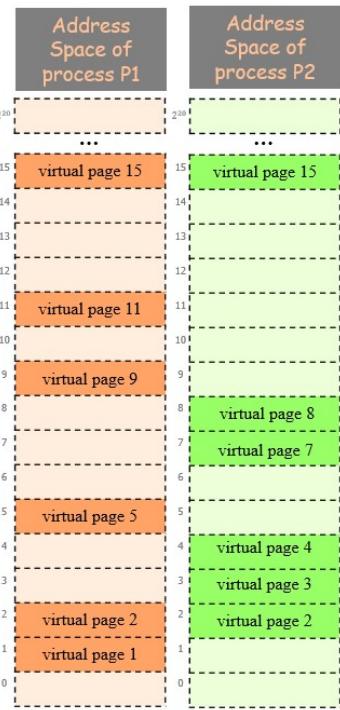
$5000 \% 4 \text{ Kb} = 5000 \% 4096 = 904 \text{ bytes}$
→ go to the byte 904 in the frame #2
→ go to the byte $2 \times 4096 + 904 = 9,096$ in RAM

→ mov EAX, [9096]



Address Space of process $2^{32} = 4Gb$





32-bit architecture
→ 2^{32} bytes Address Space

4 Kb = 2^{12} bytes page size
→ $2^{32} / 2^{12} = 2^{20}$ pages in Address Space
→ 2^{20} entries in Page Table

PTE (Page Table Entry) keeps address of frame in RAM. So, **PTE size** is (at least) 4 bytes.

PTE (Page Table Entry) keeps address of frame in RAM
→ PTE size is 4 bytes
→ **Page Table size is:** 2^{20} entries * 4 bytes per PTE = **2²² bytes = 4 Mb**

64-bit architecture
→ 2^{64} bytes Address Space

4 Kb = 2^{12} bytes page size
→ $2^{64} / 2^{12} = 2^{52}$ pages in Address Space
→ 2^{52} entries in Page Table

What about 64-bit architecture?

PTE (Page Table Entry) keeps address of frame in RAM
→ PTE size is 4 bytes
→ **Page Table size is:** 2^{52} entries * 4 bytes per PTE = **2⁵⁴ bytes = 4096 Tb**

We cannot keep the whole Page Table in RAM. What can be done?

Let's page the Page Table!

Page Table of P1

virtual addresses range	virtual page number	physical frame number	present/absent bit
3.9G-4G	2^{20}	000	0
...			
60K-64K	15	010	1
56K-60K	14	000	0
52K-56K	13	000	0
48K-52K	12	000	0
44K-48K	11	111	1
40K-44K	10	000	0
36K-40K	9	000	0
32K-36K	8	000	0
28K-32K	7	000	0
24K-28K	6	000	0
20K-24K	5	000	0
16K-20K	4	000	0
12K-16K	3	000	0
8K-12K	2	110	1
4K-8K	1	001	1
0K-4K	0	000	0

באו נתחילה בדוגמה קונקרטית. נניח שיש לנו זיכרון וירטואלי בנפח של 4GB. זה משמע שיש לנו 2^{20} דפים, מאחר וזהו התוצאה של חלוקת 4GB ב-4KB. זו תמונה יותר רלוונטית למורדות שבימינו המעבדים הם בעלי 64 ביטים, ולכן יהיו לנו 2^{52} דפים.

נניח שהמעבד רוצה לבצע את הפקודה הבאה: [5000 mov EAX, 5000]. זו הוראה שambilketת לטען את התוכן שנמצא בכתובת 5000 לטור הרגיסטר EAX. הכתובת 5000 היא כתובות וירטואלית, והמטרה שלנו היא למצוא את הכתובת הפיזית אליה היא מתייחסת.

כדי למצוא את הכתובת הפיזית, אנחנו צריכים לעבור שני שלבים. שלב הראשון הוא למצוא את הדף המתאים בטבלת הדפים. נחלק את הכתובת הווירטואלית בגודל הדף וניקח את החלק השלים. מאחר גודל הדף הוא 4096 בתים, תוצאה החלוקת היא 1. כמובן, אנחנו צריכים לאשת לדף הווירטואלי מס' 1.

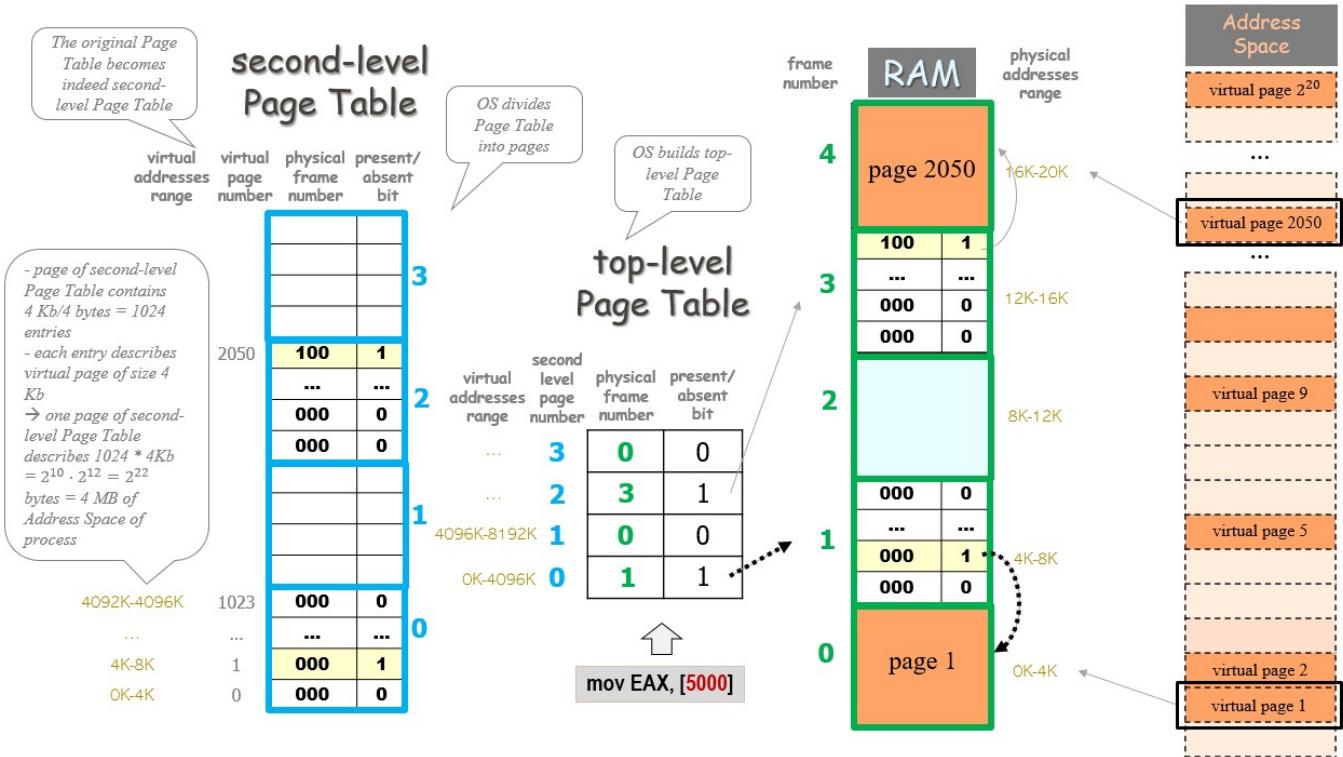
לפי הרשומה בטבלת הדפים, הדף הווירטואלי מס' 1 נמצא במסגרת הפיזית מס' 2. כעת, כדי לקבל את הכתובת הפיזית, אנחנו מחשבים את השארית של חלוקת 5000 ב-4096, שהוא 904, ומוסיפים אותה לתחילת הכתובת הפיזית במסגרת זו. לכן, הכתובת הפיזית האמיתית היא $2^{12} \times 4096 + 904$, שהוא 9096.

החסרון הבולט במערכת זו הוא העלות הגבוהה של מספר הגישות לזכרון. יש לנו גישה אחת למטען ההוראה, שתיים נוספות לארגומנטים, שתים נוספות לטבלת הדפים, וגישה נוספת לכתיבה או לקריאה.

שימוש נוסף בדפים הוא עבור שיתוף זיכרון. נזכר בקריאה mmap למיפוי משותף של הזיכרון. כדי לתמוך זה באסטרטגייה של דפים, אפשר להגדיר שהדפים הווירטואליים יצביעו לאותו מסגרת. זאת הגדסה הכללית של ספריות משותפות.

נקודה נוספת שחייב לזכור היא שמערכת הפעלה בונה את הטבלאות הללו מבני נתונים. לשם כך, מערכת הפעלה חייבת להזכיר מקום ברם לשמרות הטבלאות עצמן. אם המעבד הוא מעבד 32 ביט, אז בשביל מרחב כתובות בגודל 2^{32} יש 2^{20} כניסה בטבלת הדפים, כאשר כל כניסה היא בגודל 4 בתים, ולכן גודל הטבלה הוא $4MB = 4 \times 2^{20}$.

זה אודול מדי, במיוחד עבור מעבדים של 64 ביט. בשביל זה נשתמש בכתובות וירטואליות עבור טבלת הדפים במילימ אחורות, נשמר טבלת דפים לטבלת דפים, בצורה היררכית.



טבאות דפים לזכרוןות גדולים

טבאות דפים מרובות שלבים

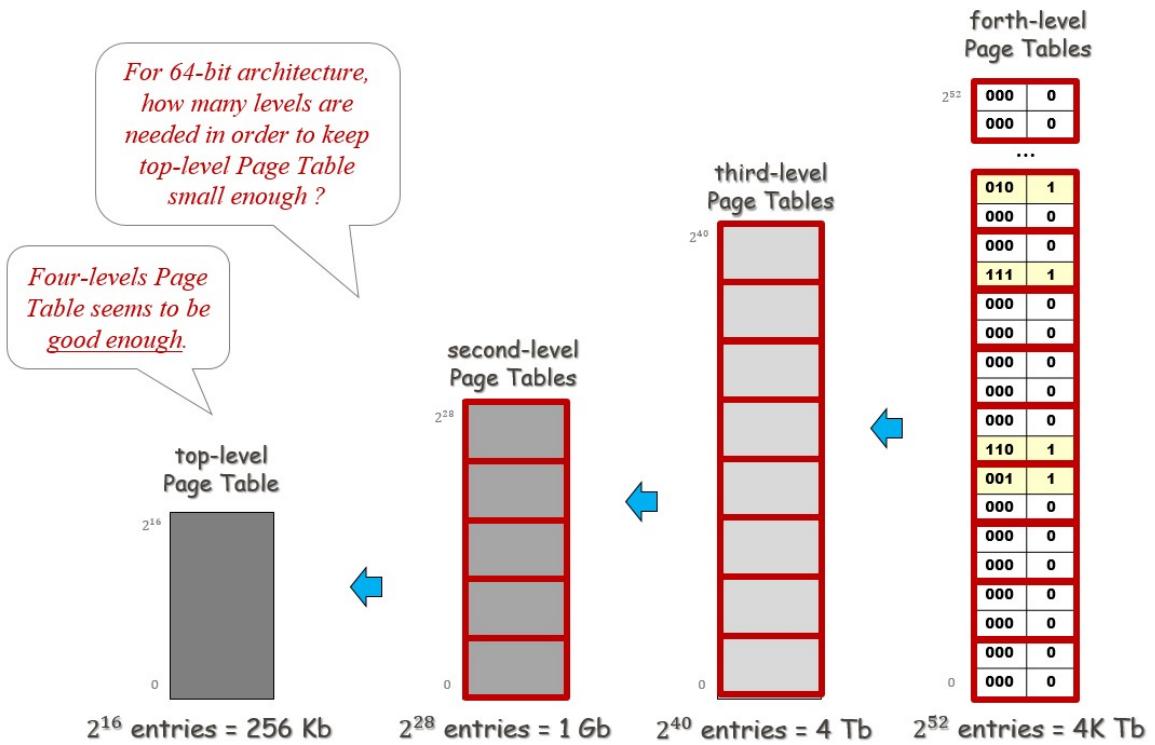
מערכת ההפעלה בונה את טבלת הדפים ברמה העליונה, ומחלקת את טבלת הדפים לדפים. הטבלה המקורית הופכת להיות טבלת דפים ברמה השנייה. בעת, יש לנו מרחב כתובות שמחולק לדפים, כמובן, יש לנו שתי טבאות: הטבלה הראשונה ממפה דפים וירטואלים לمسגרות פיזיות, והטבלה השנייה ממפה כתובות וירטואלית למיקום המתאים בטבלה הראשונה.

אם נחזור לדוגמה שלנו מקודם,icut, כדי לפענח את הכתובת הווירטואלית 500, נחפש בטבלה הראשית (ברמה העליונה) את המיקום ברמה התחתונה ומשם נחלץ את המסגרת הפיזית. החסרון העיקרי בשיטה זו הוא שהוא מוסיף טבלת דפים נוספת שאוררת אליה גישות נוספת לראם, דבר שמאט את ריצת התוכנית.

נניח שיש לנו זיכרון בגודל של 2^{32} . אז יש לנו 2^{20} דפים, וגודל הטבלה הוא 4MB. לכן, גודל הטבלה העליונה הוא 256 בתים. בארכיטקטורת 64 ביט, נדרש לחלק עוד יותר כדי להגיע לטבלה בגודל 256, וזה יגרור עוד גישות לראמ.

בארכיטקטורת 64 ביט, כמה שלבים נדרשים כדי לשמר על גודל טבלת הדפים ברמה העליונה קטן מספיק? נראה שטבלת דפים בארכעה שלבים תיהה מספקת. יש קשר בין גודל הדף לגודל טבלת הדפים: דפים קטנים יובילו לטבלת דפים קטנה יותר, אך יגרמו לפיצול פנימי אבוי (בדפים שאינם מאוכלסים כלוטין), בעוד דפים קטנים יובילו לטבלת דפים גדולה יותר (שתצריך יותר זיכרון RAM), ויבילו ליותר שגיאות דף.

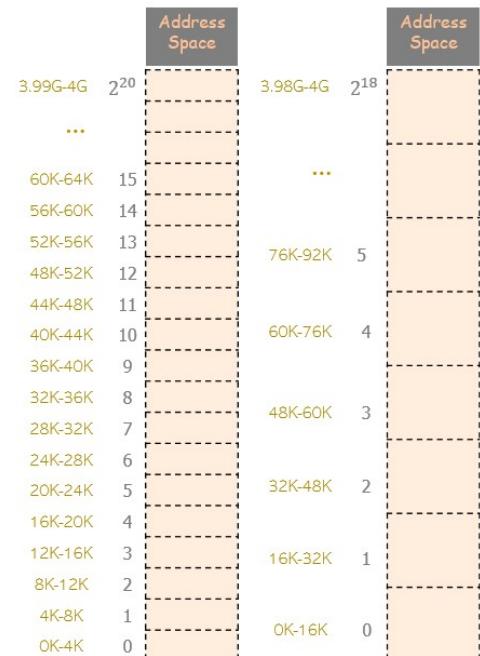
לדוגמה, אם יש לנו מרחב כתובות של 64KB, זיכרון RAM של 32KB, וגודל דף של 4KB, נדרש 12 סיביות לייצוג היחסט בדף (מספר הבית), 4 סיביות לייצוג מספר הדף (מאחר ויש לנו 16 דפים וירטואליים), ו-3 סיביות לייצוג מספר המסגרת (מאחר ויש לנו 8 מסגרות).



Page size vs. Page Table size tradeoff

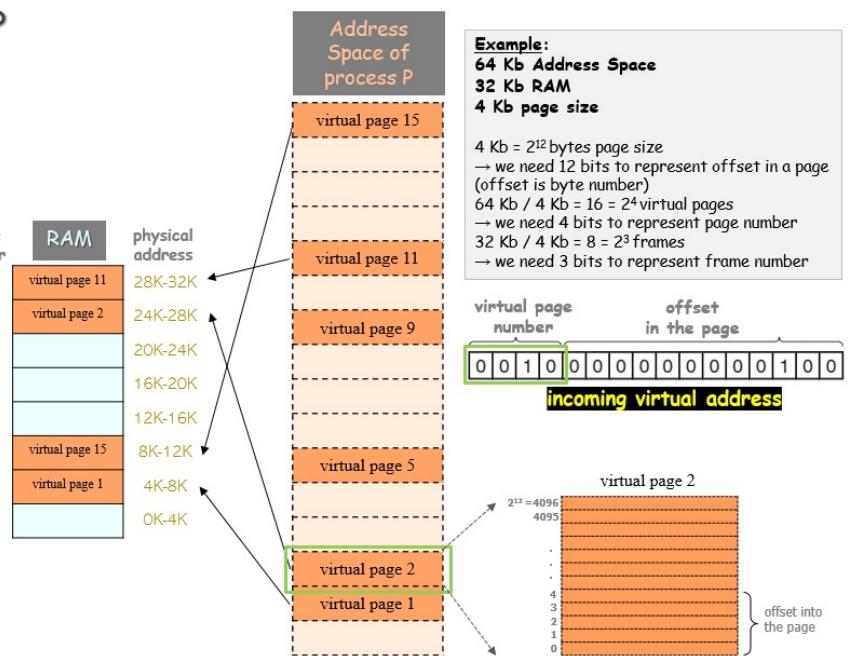
Page Table is per-process structure

- A 32-bit Address Space may be divided into:
 - 4K sized pages => 2^{32} bytes / 4 Kb = 1M pages
 - 16K sized pages => $\frac{1}{4}$ M pages
 - ...
- **Large pages** – smaller Page Table, but high internal fragmentation (for not fully occupied pages)
- **Small pages** – larger Page Tables (need more RAM for it), more page faults



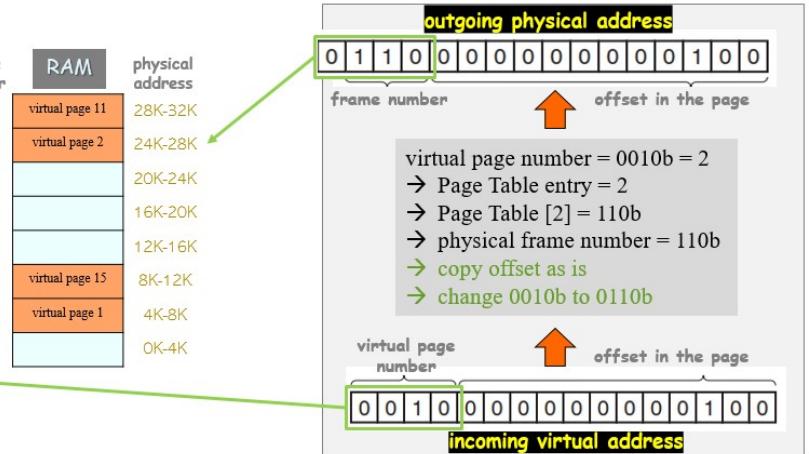
Page Table of process P

virtual addresses range	virtual page number	physical frame number	present/ absent bit
60K-64K	15	010	1
56K-60K	14	000	0
52K-56K	13	000	0
48K-52K	12	000	0
44K-48K	11	111	1
40K-44K	10	000	0
36K-40K	9	000	0
32K-36K	8	000	0
28K-32K	7	000	0
24K-28K	6	000	0
20K-24K	5	000	0
16K-20K	4	000	0
12K-16K	3	000	0
8K-12K	2	110	1
4K-8K	1	001	1
0K-4K	0	000	0



Page Table of process P

virtual addresses range	virtual page number	physical frame number	present/ absent bit
60K-64K	15	010	1
56K-60K	14	000	0
52K-56K	13	000	0
48K-52K	12	000	0
44K-48K	11	111	1
40K-44K	10	000	0
36K-40K	9	000	0
32K-36K	8	000	0
28K-32K	7	000	0
24K-28K	6	000	0
20K-24K	5	000	0
16K-20K	4	000	0
12K-16K	3	000	0
8K-12K	2	110	1
4K-8K	1	001	1
0K-4K	0	000	0



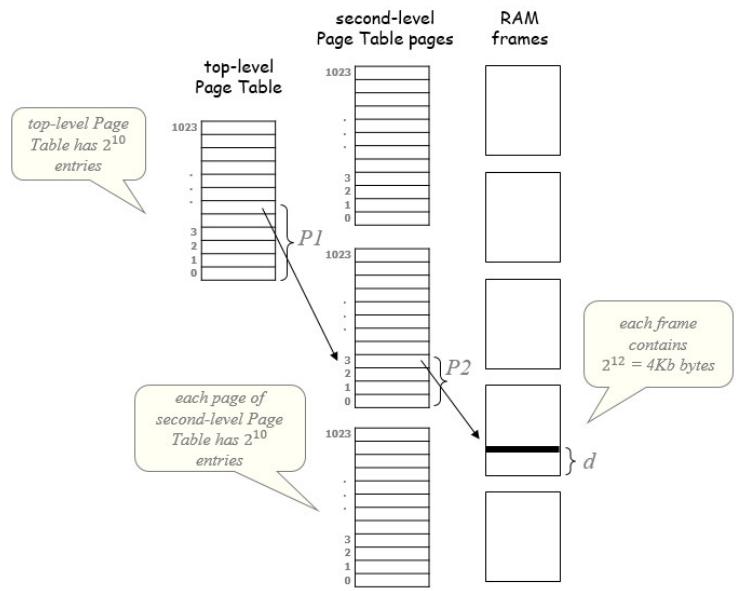
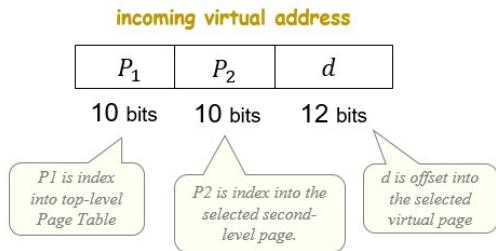
Two-Level Page Table Scheme

32-bit architecture
 $\rightarrow 2^{32}$ bytes Address Space

$4 \text{ Kb} = 2^{12}$ bytes page size
 $\rightarrow 4 \text{ Kb} / 4 \text{ bytes entry size} = 2^{10}$ entries in each page of second-level Page Table
 $\rightarrow 2^{32} / 2^{12} = 2^{20}$ pages in Address Space
 $\rightarrow 2^{20}$ entries in Page Table

PTE keeps address of frame in RAM
 \rightarrow PTE size is 4 bytes
 \rightarrow Page Table size is: 2^{20} entries * 4 bytes per PTE = 2^{22} bytes

Divide the Page Table into 4 Kb-sized pages
 $\rightarrow 2^{22}$ bytes / 4 Kb = 2^{10} second-level pages
 \rightarrow we need 2^{10} entries in the top-level of the Page Table



לפני שנדבר על הפתרון (שייה באמצעות מטמון, לא של המעבד), נדבר על אודל הדף. אם הגודל הוא 4KB, אז הדפים הם בגודל 4KB, וזה יהיה לנו יותר טבלאות. נראה שאודל דף של 16KB יהיה טוב יותר, אך אז אנו ממלאים את הרם מהר מדי, ואני יכולם להגיד שהוא מחייב מעלים את כל תמונה התהילה לרם.

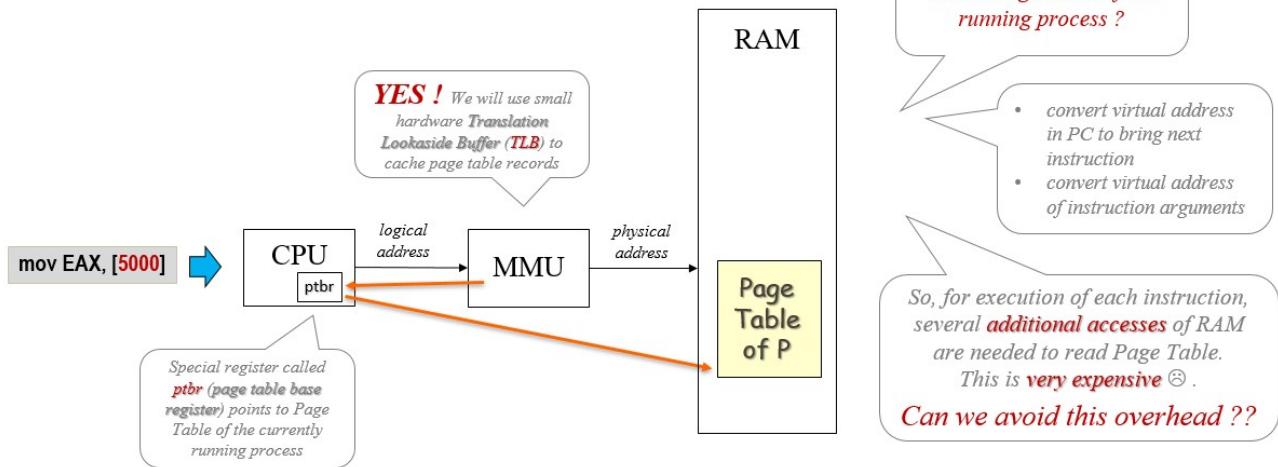
לנו יש טבלת דפים ראשונית וכ כתובת וירטואלית. הרעיון הוא שהבטים הראשונים מייצגים באיזה דף הוא נמצא והשאר הם המיקום מתחילה הדף. זה אומר שכאשר המעבד רוצה ל千古ת מה כתובת, הוא פשוט יctrkr להחליף את מספר הדף במספר המסגרת.

הסוד של שיטת טבלת הדפים ברמות מרובות הוא להימנע משמירה על כל טבלאות הדפים בזיכרון כל הזמן. במיוחד, אלה שאינם נדרשים לא צריכים להישמר. לדוגמה, נניח שתהיליך זוקק ל-12 מגה-בייט: 4 מגה-בייט התחthonים של הזיכרון לטקסט התוכנית, 4 מגה-בייט הבאים לננתונים, ו-4 מגה-בייט העליונים למחסנית. בין ראש הנתונים לתחתית המחסנית יש חור עצום שאינו משמש.

באמציאות טבלת הדפים ברמות מרובות, אנו יכולים לנצל את מרחב הכתובות באופןיעיל יותר, תוך שמירה על מספר מינימלי של טבלאות דפים בזיכרון.

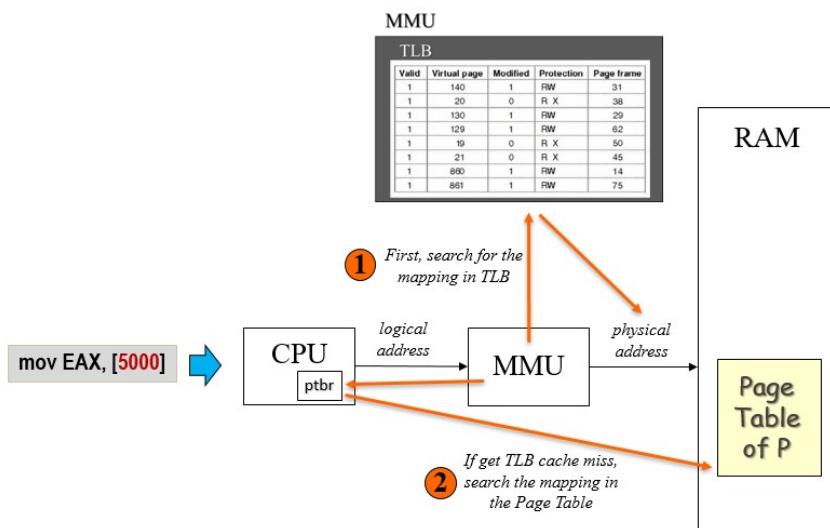
Memory Management Unit (MMU)

- MMU converts logical (virtual) address to physical address
 - each conversion requires access to the Page Table of the running process



Translation Lookaside Buffer (TLB)

associative cache for minimizing redundant memory accesses

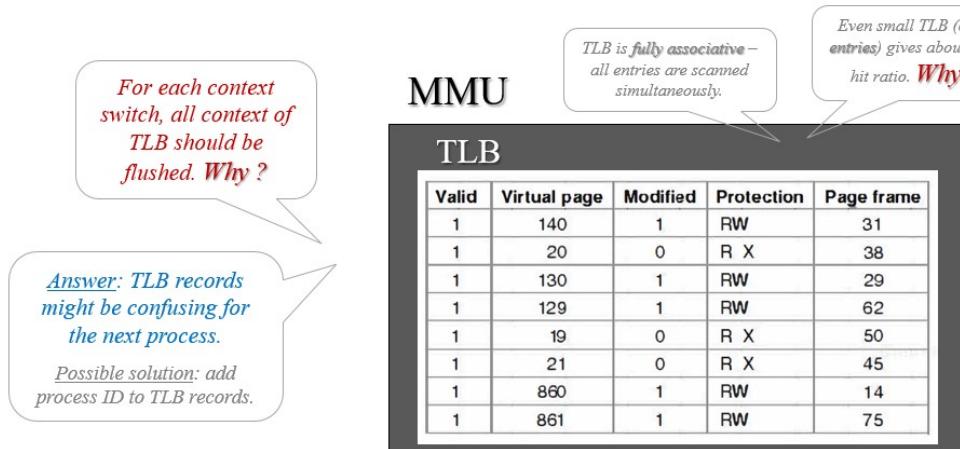


Translation Lookaside Buffer (TLB)

associative cache for minimizing redundant memory accesses

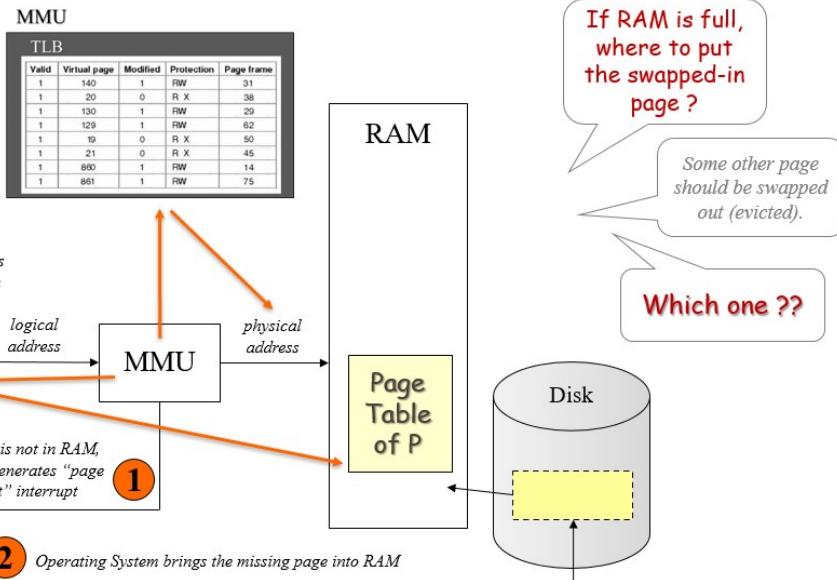
Locality of reference:

- most accesses of a process are to a small subset of pages
- by using even small TLB we get **high hit ratio**



MMU - Page Fault

- if MMU does not find the needed page in RAM
 - MMU produces "page fault" interrupt
 - OS blocks the process, and brings the needed page into RAM
 - CPU completes the instruction execution



יחידת ניהול זיכרון (MMU)

יחידת ניהול זיכרון, או MMU, היא רכיב חומרה שממיר כתובות לוגית (וירטואלית) לכתובת פיזית. כל המריה כזו דורשת אישה לטבלת הדפים של התהילך המתאים. התדריות שבה המעבד צריך לגשת לטבלת הדפים של התהילך תלויה במספר אורומים, כולל הצורך להמיר כתובות וירטואלית במחשב האישי להבראה הbara, והצורך להמיר כתובות וירטואלית של ארגומנטים של הוראה. לכן, לביצוע כל הוראה, נדרש מספר גישות נוספת ל-RAM כדי לקרוא את טבלת הדפים. זה יכול להיות יקר מבחינת ביצועים.

או, האם אפשר להימנע מהעומס הזה? התשובה היא כן! אנחנו יכולים להשתמש במתגן חומרה קטן, הנקרא Translation Lookaside Buffer (TLB) שמאפשר רשותות של טבלת הדפים בסמוך למעבד. ישנו רגיסטר מיוחד שנקרא ptbr (page table base register) שמצויב לטבלת הדפים של התהילך המרואה כרגע.

Translation Lookaside Buffer (TLB)

ה-TLB הוא מטמון אסוציאטיבי שמטרתו לסייע גישות זיכרון מיותרות. הוא מבוסס על העיקנון של "reference", שאומר שרוב הגישות של תהליך הן לתת-קובוצה קטנה של דפים. על ידי שימוש ב-TLB קטן אפשר להשיג יחס פגיעה גדול יותר כדי שימוש ב-TLB קטן. ה-TLB הוא מטמון אסוציאטיבי מלא - כל הרשומות נסרקות במקביל. עבור כל החלפת הקשר, כל ההקשר של ה-TLB צריך להימחק, כי רשותות TLB עשויות להיות מבלבלות במקרה. עבור התהילך הבא.

אינטואיטיבית: קל לנו לשנן את המיקום של הפריטים שהשימוש בהם הוא כי תדייר, אז נזכיר בראש רשימה קטנה של פריטים שהם בשימוש תדייר.

התהילך של ה-TLB הוא כזה: תחילה, מחפשים את המיפוי ב-TLB. אם מתקבל החטאה במטמון של ה-TLB, מחפשים את המיפוי בטבלת הדפים. אם ה-MMU לא מוצא את הדף הנדרש ב-RAM, ה-U-MMU מייצר פסיקה של "page fault". מערכת הפעלה חוסמת את התהילך, וمبיאה את הדף הנדרש ל-RAM. המעבד משלים את ביצועה ההוראה. אם ה-RAM מלא, איפה לשים את הדף שהוחלף? דף אחר צריך להיחלף (להיגרש). איזה מהם? נראה אלגוריתמי החלפת דפים בהמשך.

ה-U-MMU משתמש בראיסטר מיוחד הנקרא Page Table Base Register אשר מצביע לטבלת הדפים של התהילך הנוכחי שרצ רכגע. במידה וה-MMU לא מוצא את הדף הדרש בראם הוא יזרוק חריגת pagefault אשר בעקבותיה מערכת הפעלה תשים את התהילך ב-block-fault.

ה-TLB משפר את הביצועים על ידי הפחיתת מספר הגישות הנדרשות לזכור לתרגם כתובות, גם עבור TLB יחסית קטן. כמו כן הוא מאפשר תרגום כתובות מהיר יותר, ובכך מקטין את הפגיעה בביצועים כתוצאה מ-page-fault.

ברגע שמתרכז החלפת הקשרים, מתבצע TLB flush כדי למנוע התנגשות בין ערכי ה-TLB של תהליכיים שונים. כמו כן הוא מאפשר ידואו שהמיפויים בזיכרון רלוונטיים לתהילך החדש שרצ.

פתרון אפשרי: הוספת מזווה התהילך לרשותות TLB.

32-bit Page Table Entry (PTE)

Page Table of process P

- Frame number (physical address) _____
- Present/absent (valid) bit _____
- Protection _____
- Dirty (modified) bit _____
- Referenced bit _____
- Caching disable/enable _____

PTE structure

virtual address	virtual page number	physical frame number	present/absent bit	
60K-64K	15	010	1	
56K-60K	14	000	0	
52K-56K	13	000	0	
48K-52K	12	000	0	
44K-48K	11	111	1	frame number
40K-44K	10	000	0	RAM
36K-40K	9	000	0	physical address
32K-36K	8	000	0	virtual page 11 28K-32K
28K-32K	7	000	0	virtual page 2 24K-28K
24K-28K	6	000	0	20K-24K
20K-24K	5	000	0	16K-20K
16K-20K	4	000	0	12K-16K
12K-16K	3	000	0	8K-12K
8K-12K	2	110	1	virtual page 15 8K-12K
4K-8K	1	001	1	4K-8K
0K-4K	0	000	0	0K-4K

Why do we need referenced bit ?

Referenced bit is used to decide which page should be evicted from RAM.

מבנה ה-PTE והאבסטרקציה של זיכרון וירטואלי.

כדי להבין את האלגוריתמים לניהול דפי זיכרון נדרש לצלול יותר פנימה. המבנה של ה-PTE והתוכן שלו משקדים תפקיד חשוב בניהול זיכרון וירטואלי. שימושם הוא למיפוי הכתובות הווירטואליות לכטבות הפיזיות, תוך כדי ייעולGISות לזיכרון וניהולו. המבנה שלו משתנה מארQUITקטורה אחת לאחרת אך התוכן של דומה.

השדה החשוב ביותר הוא מספר Page Frame אשר מייצג את הכתובת הפיזית המתאימה לדף הווירטואלי.

ביט ה-Present/Absent מאפשר לדעת האם הרשמה היא תקפה והאם היא בזיכרון. ערך 0 של בית זה גורר שpage.fault

ה-**Protection bits** קובעים את הרשות הגישה לדף, בדרך כלל מוחבר בהרשאות קריאה, כתיבה והרצה.

ה-**bit Dirty** או ה-Bit (Modified) נدلך כאשר התרחש שינוי בדף. נועד לעקוב אחר המצב של הדף והאם יש צורך לכתוב אותו חזרה לראמ או לדיסק. מdad טוב כדי לדעת האם ההורדה מראם היא יקרה לא יקרה.

ה-**Referenced bit** חשוב לאלגוריתמים להחלפת דפים כאשר יש fault.page. הוא עוקב אחר השימוש בדף על ידי עדכנו בכל פעם שהייתה גישה לדף. בעזרת הביט הזה מערכת הפעלה יכולה לשקל האם הדף "אטרטיבי" מבחינתה - ככל שהbeit שלו דולק יותר ומן, נראה שהוא יותר בשימוש.

ביט ה-Caching Disable/Enable מאפשר להפסיק לשמר במתמון דפים מסוימים. קריטי עבר דפים אשר ממפים רגיסטרים של התקנים במקום זיכרון ולתהליכיים ספציפיים כמו מערכת הפעלה.

סיכום מודל טבלת הדפים הרגילה:

הזיכרון הראשי - מחולק למסגרות (או דפים פיזיים).

תמונה הזיכרון של התהיליך - מחולק לדפים וירטואליים.

בכל פעם טענים למסגרת פיזית דף וירטואלי אחד. ניהול מקום הדפים בזיכרון הראשי נעשה בעזרת טבלת הדפים. מספר הרשותות בטבלת הדפים הראשית הוא כמספר הדפים שתהilih מקבל. הרשותות בטבלה הן PTE, שמקילות ביטים לצרכים שונים וקידוד של מקום הפריים הפיזיים.

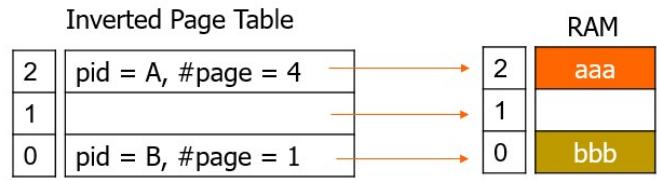
אם הטבלה אדולה מדי אפשר לחלק אותה לטבלאות קטנות יותר אז חלק מהטבלאות יהיו בראם וחלק יהיו בדיסק.

Inverted Page Table

a single table that maps physical frames to virtual pages

Motivation: for 64-bit address space, page table size is 2^{52} pages x 8 bytes = 4K TB

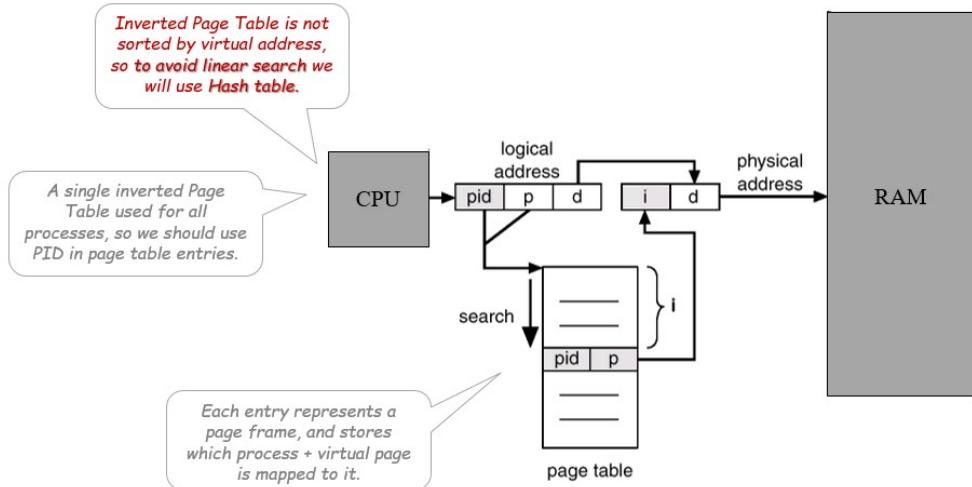
process A Page Table		process B Page Table	
6		6	
5		5	
4	aaa	4	
3		3	
2		2	
1		1	bbb
0		0	



Inverted Page Table

a single table that maps physical frames to virtual pages

Motivation: for 64-bit address space, page table size is 2^{52} pages x 8 bytes = 4K TB



טבלת דפים הפוכה

טבלת דפים הפוכה היא גישה אלטרנטיבית להיררכיות הדפים המסורתיות בניהול זיכרון. במקום להכיל רשומה אחת לכל דף במרחב הכתובות הווירטואליות, טבלת הדפים ההפוכה מכילה רשומה אחת לכל פריים דף בזיכרון הפיזי. זה מאפשר חיסכון משמעותי במקום, במיוחד מרחיב הכתובות הווירטואליות גדול לעומת הזיכרון הפיזי. לדוגמה, עבור מרחב כתובות בגודל 64 ביט, גודל טבלת הדפים יכול להגיע ל-4 קילובייט.

טבלת הדפים ההפוכה אינה ממיינת לפי הכתובת הווירטואלית, ולכן, כדי למנוע חיפוש לנינאי, אנו משתמשים בטבלת האש. טבלת הדפים ההפוכה משמשת לכל התהליכיים, ולכן, אנו צריכים להשתמש במחזה התהיליך (PID) ברשומות טבלת הדפים. כל רשומה מייצגת פריים דף, ומאחסנת איזה תהיליך ודף וירטואלי ממופים אליו.

אף על פי שטבלת הדפים ההפוכה חוסכת הרבה מקום, יש לה חסרן שימושו: התרגומם מ כתובות וירטואליות לפיזית הופך להיות מאוד מורכב. כאשר תהליך מפנה לדף וירטואלי, החומרה אינה יכולה למצוא את הדף הפיזי עלי ידי שימוש בדף הוירטואלי כאינדקס בטבלת הדפים. במקרה זאת, היא חייבת לחפש את כל טבלת הדפים ההפוכה עבור רשותה שמכילה את התהילך והדף הוירטואלי. חיפוש זה צריך להתבצע בכל פניה ליצירון, ולא רק בעת חריגת דף, דבר שפגיע בביטויים.

דרך להימנע עם הבעיה זו היא להשתמש ב-TLB. אם ה-TLB יכול לשמר את כל הדפים שימושיים באופן קבוע, התרגומם יוכל להתבצע באותה מהירות כמו עם טבלאות דפים וגילות. עם זאת, במקרה של חטאה ב-TLB, יש לחפש את טבלת הדפים ההפוכה בתוכנה. דרך אחרת לבצע חיפוש זה היא להשתמש בטבלת האש שמשתמשת בכתובות וירטואליות כמפתחות. כל הדפים הוירטואליים שנמצאים כרגע בזיכרון יש להם ערך האש משורשרים ייחד. בצורה זו, אם יש לטבלת האש מספר תאים שווה למספר הדפים הפיזיים, אורך שרשרת הממוצע יהיה רק רשותה אחת, מה שהיא באופן שימושי את תהליך המיפוי.

למרות היתרון של טבלת הדפים ההפוכה, היא הגיעה מאוחר מדי ולא התקבלה באופן נרחב. המנגנון המסורתית עובד טוב, ולכן אין צורך להשקיע זמן וכסף בשינויו.

בספר של טננבראום (מהדורה 5), טננבראום מציגشرطו שונה של טבלת הדפים ההפוכה, הוסיף אותה כי לדעתו היא מוגנת יותר:

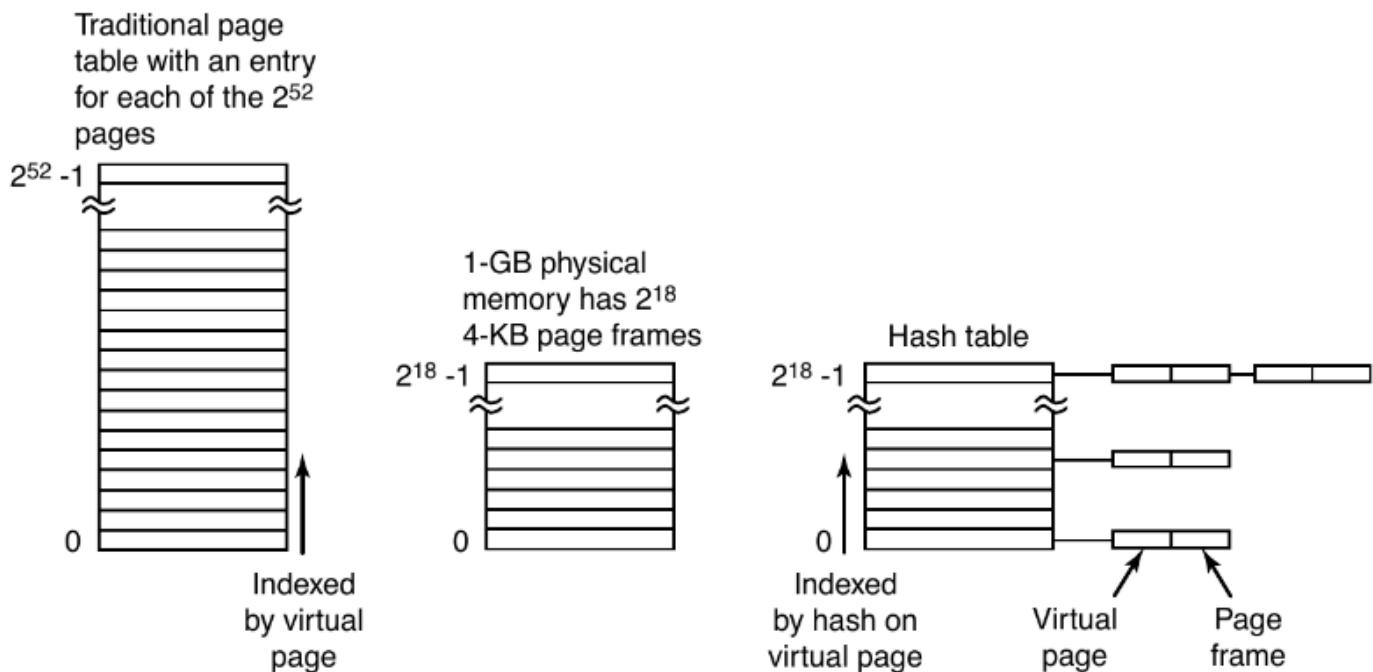


Figure 3-14. Comparison of a traditional page table with an inverted page table.



© 2022 מילוט הפעלה

Operating Systems

Lecture 8 – Memory Management – part 2

Dr. Marina Kogan-Sadetsky

Course Syllabus

- 1. Introduction**
- 2. Process Management**
- 3. Scheduling algorithms**
- 4. Synchronization**
- 5. Memory Management**
 - Page replacement algorithms
 - Working set model
 - Virtual memory implementation issues
- 6. File Systems**
- 7. Virtualization**



Memory management: outline

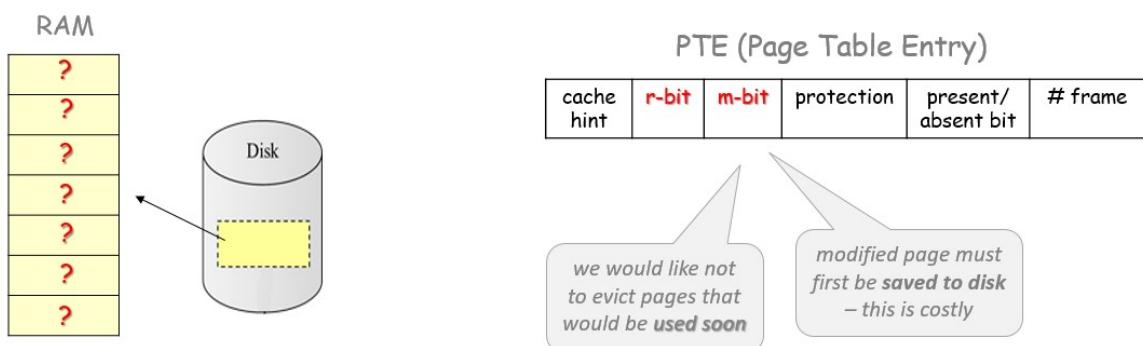
- ❑ Page replacement algorithms
- ❑ Working set model
- ❑ Virtual memory implementation issues

Page Replacement Algorithms

כאשר הזיכרון הפיזי מלא, מערכת הפעלה חייבת לבחור את הדף אשר יש להוריד מהראמ על מנת לפנות מקום לדפים נוספים. הבחירה של איזה דף להוריד היא קריטית עבור הביצועים של המערכת והיעילות שלה. הבחירה של איזה דף להוריד היא בחרה חמדנית. האידיאל אומר להוריד דפים שאין בהם שימוש תדיר. אם הדף השתנה בזמן היותו בזיכרון, הוא חייב להיות מוחזר לדיסק כדי לעדכן את העותק בדיסק. אם הדף לא השתנה, אין צורך בשכתובים שכן העותק שבדיסק בכל מקרה מעודכן.

If physical memory is full, some page should be evicted.

Which one ??



החשיבות של אלגוריתמי החלפת דפים:
בחירה רנדומלית של דפים בכל page fault אפשרית אך מובילת לפגיעה בביצועים של המערכת. לשם כך על

האלגוריתמים להתחשב בתדריות השימוש של הדף כדי לספק תוצאות טובות יותר. יתר על כן על בחירת הדף להיות מהירה ובכלי הרצן הוא שהאלגוריתמים ירצו מהר ולכן הם יהיו פשוטים.

בנוסף סוגיה נוספת שהאלגוריתמים צריכים להתמודד איתה היא האם הדף שהורד שייר לתוכהו שבעקבותיו הגיעת-h-page fault או של תחילך אחר.

הפתרון האולטימטיבי הוא בעצם לפנות את ה-Page שהשימוש הבא שלו הוא הכி מאוחר, אך אין למערכת הפעלה מושג איך לדעת זאת ולכן אין מימוש שלו.

OPT - optimal page replacement algorithm

What is it ?
Any ideas ?

The optimal Page Replacement Algorithm

בתיאוריה הוא האלגוריתם הכי טוב אך לא ניתן למימוש פרקטי. הכלל החמדני של האלגוריתם הוא מחיקת הדף שהאזכור הבא שלו הוא הכி רחוק בעtid. בשביל לכך האלגוריתם חייב לדעת על דרישות עתידיות לדפים מצד התהיליך, דבר שהוא לא ריאליסטי.

OPT - optimal page replacement algorithm

- Remove the page that will be referenced [latest in the future](#)
- Unrealistic: OPT requires knowledge of future requests

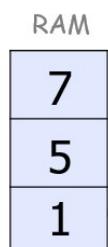
page requests sequence: **0 5 4 7 0 2 1 0 7**



OPT - optimal page replacement algorithm

- Remove the page that will be referenced [latest in the future](#)
- Unrealistic: OPT requires knowledge of future requests

page requests sequence: **0 5 4 7 0 2 1 0 7**



בhinintן הסדרה לעיל ובהנחה ודף 7, 5, 1 נמצאים בזיכרון.
אדום - החטאה

0 לא ברם אז צריך לבצע החלפה. לפי כלל הבחירה נעיף את 1.
5 נמצא בזיכרון
4 - החטאה ולכן נוציא את 5 (לא צריך אותו יותר).
וכך להלאה...

יש כאן מהו שנואגד את הרעיון של bit referenced, רעיון שבו הרבה אלגוריתמים כן משתמשים. זהו אחד הבודדים שלא משתמשים בו.

כאמור, הוא אלגוריתם תיאורטי ואין יישום באף מערכת הפעלה. אנחנו נשתמש בו על מנת להשוות תוצאות של אלגוריתם שהוכח כעובד ביחס לאלגוריתמים אחרים. אנחנו כל פעם נחשב על הסדרה "שהכי" מפילה את האלגוריתם שאנו חושבים עליו ונשווה אותו ביחס לאלגוריתם הזה.

OPT - optimal page replacement algorithm

- Remove the page that will be referenced [latest in the future](#)
- Unrealistic: OPT requires knowledge of future requests

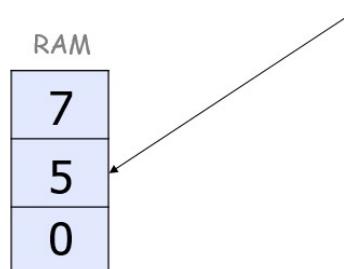
page requests sequence: **0 5 4 7 0 2 1 0 7**



OPT - optimal page replacement algorithm

- Remove the page that will be referenced [latest in the future](#)
- Unrealistic: OPT requires knowledge of future requests

page requests sequence: **0 5 4 7 0 2 1 0 7**



OPT - optimal page replacement algorithm

- Remove the page that will be referenced [latest in the future](#)
- Unrealistic: OPT requires knowledge of future requests

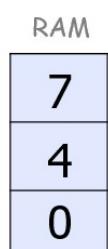
page requests sequence: **0 5 4 7 0 2 1 0 7**



OPT - optimal page replacement algorithm

- Remove the page that will be referenced [latest in the future](#)
- Unrealistic: OPT requires knowledge of future requests

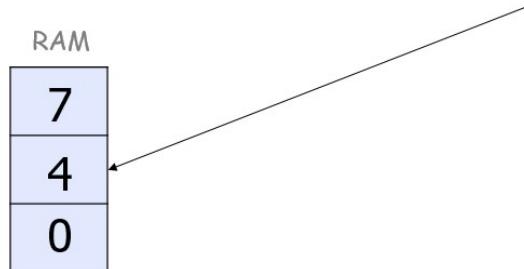
page requests sequence: **0 5 4 7 0 2 1 0 7**



OPT - optimal page replacement algorithm

- Remove the page that will be referenced [latest in the future](#)
- Unrealistic: OPT requires knowledge of future requests

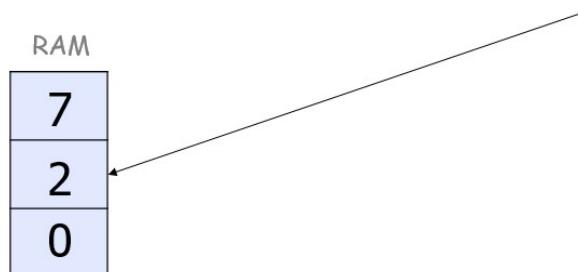
page requests sequence: 0 5 4 7 0 2 1 0 7



OPT - optimal page replacement algorithm

- Remove the page that will be referenced [latest in the future](#)
- Unrealistic: OPT requires knowledge of future requests

page requests sequence: 0 5 4 7 0 2 1 0 7



OPT - optimal page replacement algorithm

- Remove the page that will be referenced [latest in the future](#)
- Unrealistic: OPT requires knowledge of future requests

page requests sequence: 0 5 4 7 0 2 1 0 7



OPT - optimal page replacement algorithm

- Remove the page that will be referenced [latest in the future](#)
- Unrealistic: OPT requires knowledge of future requests

page requests sequence: 0 5 4 7 0 2 1 0 7



Altogether 4 page replacements.
This is the best result we can get for the
given sequence.

We will use this result for
comparing realistic algorithms.

So which algorithm to use ? Maybe FIFO ?

לפי האלגוריתם ערך פתרון אופטימלי הוא 4.

FIFO as replacement algorithm

replace the oldest page

- Remove the page that was inserted first into RAM

assume that 7 was inserted first, then 5, and then 1

page requests sequence: 0 5 4 7 0 2 1 0 7



FIFO

אלגוריתם מבוסס FIFO הוא אלגוריתם עם overhead נמוך אשר מוחק את הדף האחרון שהוכנס לראם בכל page fault.

לצורך המחשה נדגים בעזרת אנלוגיה לסופרמרקט.

נניח כי בסופר עם מספר מוגבל של מקומות במדפים, ניתן להציג לכל היותר K פריטים שונים. כאשר פריט חדש נקלט בסופר, הוא מחליף אותו בפריט ישן בשביל שייהי לו מקום.

דרך אחת להסרת המוצר הוא להסיר את המוצר שהוא במלאי הכיל הרבה זמן, על סמך סדר קליטתו בחנות. הסופר מחזיר רשותה מקושרת של המוצרים אשר נמכרים כרגע, כאשר המוצר החדש ביותר נמצא בסוף הרשימה והישן ביותר נמצא בחזות הסופר.

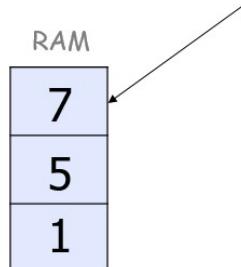
הבעיה היא שהפריט הכיל ישן בסופר הוא לא בהכרח הפריט שאינו שמיש כלל. מתוך אנלוגיה זו ניתן לפסול את FIFO שהיא הייתה אותה הסיבה שבגינה פסלו אותנו בפרק של תזמון.

FIFO as replacement algorithm

- Remove the page that was inserted first into RAM

assume that 7 was inserted first, then 5, and then 1

page requests sequence: **0 5 4 7 0 2 1 0 7**



מהלך הריצה: כאשר יש page fault, הדף בראש הרשימה (הישן ביותר) מוסר והדף החדש נכנס בתור זנב הרשימה.

ניקח את אותה הסדרה כאשר אנחנו יודעים שערך אופטימום הוא 4.
הסדר בראם: העליון - נכנס ראשון. התיכון - נכנס אחרון.

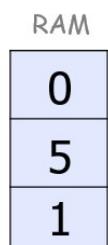
כפי שניתנו לראותות בדוגמת הריצה בסוף זרקנו את 7 בשבייל להכנס את 0 ואז הכנסנו 7 שוב. זה התרחש שמאיף את FIFO. הוא נפל כי הוא לא מתחשב בכללם. יכול להיות שלקחנו משהו שהוא לא שימושי לנו אך הורדנו דף שחשוב חשוב לנו.

FIFO as replacement algorithm

- Remove the page that was inserted first into RAM

assume that 7 was inserted first, then 5, and then 1

page requests sequence: **0 5 4 7 0 2 1 0 7**



FIFO as replacement algorithm

- Remove the page that was inserted first into RAM

assume that 7 was inserted first, then 5, and then 1
page requests sequence: **0 5 4 7 0 2 1 0 7**



FIFO as replacement algorithm

- Remove the page that was inserted first into RAM

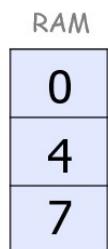
assume that 7 was inserted first, then 5, and then 1
page requests sequence: **0 5 4 7 0 2 1 0 7**



FIFO as replacement algorithm

- Remove the page that was inserted first into RAM

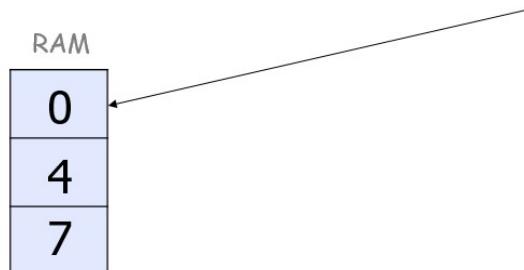
assume that 7 was inserted first, then 5, and then 1
page requests sequence: **0 5 4 7 0 2 1 0 7**



FIFO as replacement algorithm

- Remove the page that was inserted first into RAM

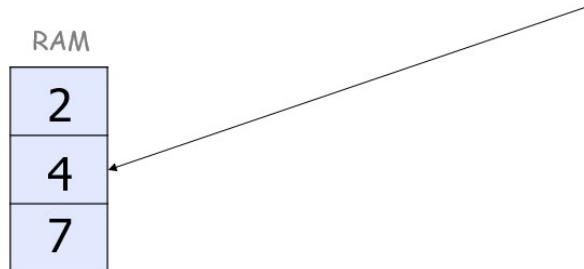
assume that 7 was inserted first, then 5, and then 1
page requests sequence: **0 5 4 7 0 2 1 0 7**



FIFO as replacement algorithm

- Remove the page that was inserted first into RAM

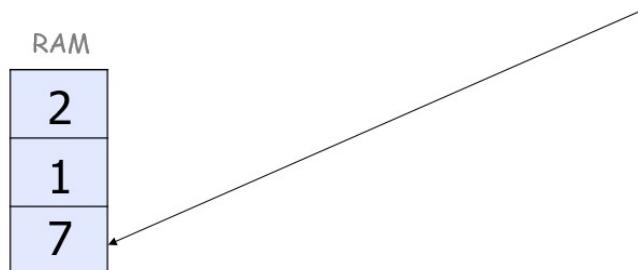
assume that 7 was inserted first, then 5, and then 1
page requests sequence: 0 5 4 7 0 2 1 0 7



FIFO as replacement algorithm

- Remove the page that was inserted first into RAM

assume that 7 was inserted first, then 5, and then 1
page requests sequence: 0 5 4 7 0 2 1 0 7



FIFO as replacement algorithm

- Remove the page that was inserted first into RAM

assume that 7 was inserted first, then 5, and then 1

page requests sequence: 0 5 4 7 0 2 1 0 7

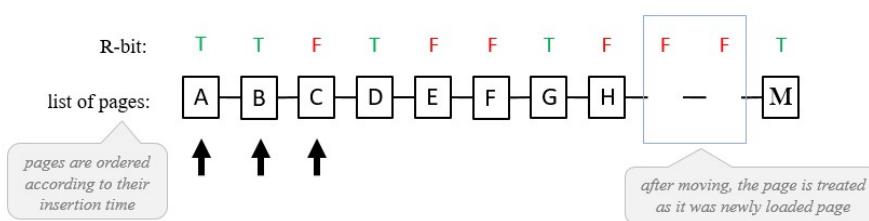


2'nd chance FIFO replacement algorithm

- inspect pages from oldest to newest
- if page's referenced bit is on, give it a second chance
 - o clear R-bit
 - o move to end of queue
- else
 - o evict the page

Note that we keep in the list only the pages that are in RAM (and not all the virtual pages of the process).

Example: page 'M' page-fault:



The Second-Chance (FIFO) Page Replacement Algorithm

מדובר בוריאציה של FIFO אשר נועדה להקטין את ההורנת דפים בשימוש תדר בראם על ידי מנגנון של הזמנות שנייה המבוסס על שימוש ב-bit referenced.

תיאור כלל הבחירה של האלגוריתם:

סדר המעבר על הדפים הוא מהישן לחידש. אם ה-bit referenced של הדף אשר נבחן ברגע זה דולק, אז כבה את בית זה והוזז את הדף לסוף התור. אחרת, הוא מוחלף בדף אחר.

הזמןות השנייה באה לידי ביטוי בכיבוי ה-bit referenced והזאת הדף לסוף התור.

דוגמאות:

בהתנחת רשות הדפים שבמצגת המסודרת לפי זמן הכנסות ובהינתן סדרת reference bit-page fault לכל דף, כאשר יש page fault עבור הדף M, הדף היישן ביותר הוא A ולכן הביט שלו נדלק והוא מוזע לסוף הרשימה, בדומה זו האלגוריתם מתיחס אליו כעדף חדש שהו吐ן וכך הלאה.

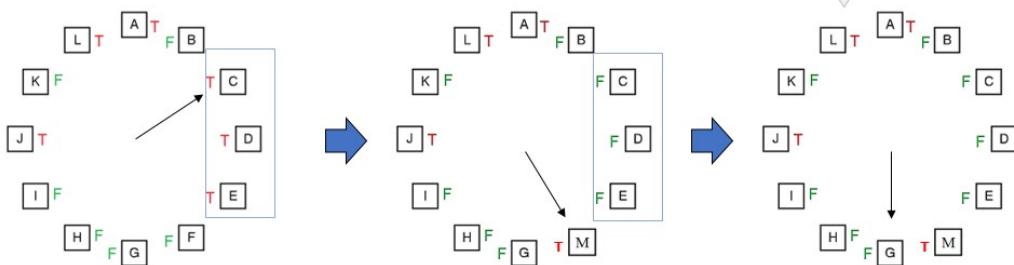
ועדיין, האלגוריתם עדין עלול לסבול מאותן הבעיה ש-FIFO סובל מהן בתרחישים מסוימים.

Clock Page Replacement Algorithm

- inspect the pointed page
- if R-bit is on
 - o clear R-bit and advance the pointer
- else
 - o evict the page

same as 2nd chance FIFO, but with circle list of pages, which leads to faster search

Example: page 'M' page-fault :



The Clock Page Replacement Algorithm

זהו שיפור של האלגוריתם הקודם אשר שומר על כל ה-pages ברשימה מעגלית המזקירה שעון.

תיאור האלגוריתם:

הdfs מסודרים בצורה מעגלית עם מעגלית עם מצביע hand אשר מצביע לדף היישן ביותר בשעון. ברגע שיש page fault הdfs מסודרים בצורה מעגלית עם מעגלית עם מצביע hand אשר מצביע לדף היישן ביותר בשעון. ברגע שיש page fault שיפור של האלגוריתם בוחן את הדף עליו hand מצביע.

בשלב זה הפעולה תלויה ב-bit referenced של הדף. אם הוא מכובה הדף מוחלף לדף אחר והמצביע hand מתקדם לחוליה הבאה. אם הוא דולק אז האלגוריתם מכבה את הביט ומקדם את hand לדף הבא.

היתרון באלגוריתם זה על פני האלגוריתם הקודם הוא בעילות זמן הריצעה - אין צורך להזיז דפים בתוך הרשימה. כמו כן רשות מעגלית מאפשרת חיפוש מהיר יותר וחלפת דפים מהירה יותר.

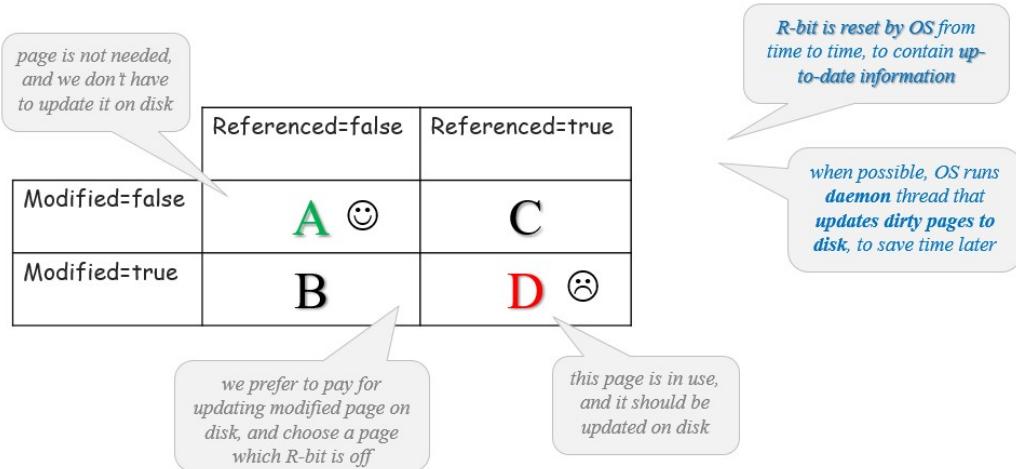
בדוגמה בשקף, כאשר יש page fault לדף M, האלגוריתם עובר על כל הדdfs עד שהוא מוצא את הראשון שה-Rbit שלו מכובה, כאשר בכל דף אחר שהוא בוחן קודם, האלגוריתם מכבה את ה-Rbit.

הערה:

צריך לזכור את האלגוריתמים בעל פה למבחן!
אנחנו בכל אלגוריתם נראה קווים מנחים לבחירות שמערכות הפעלה אמיתיות עושות.

NRU - Not Recently Used replacement algorithm

- There are four classes of pages, according to M-bit and R-bit
- Evict random page from the **least-needed class**



The Not Recently Used (NRU) Page Replacement Algorithm

זהו אלגוריתם המתבסס על הביט ה-referenced ועל ביט ה-m של כל דף ומסווג את הדפים ל-4 קטגוריות שונות לפי ערכי ביטים אלו. פונקציית המחיר כאן היא צאת שם אם modified וגם referenced דלוקות או התהיליך יקר מידית בשבייל החלפה.

תיאור האלגוריתם:

בעלית התהיליך, הביטים של הדפים מאותחלים להיות 0. מעת לעת ביט ה-R מתנתקה על מנת להבחן בין דפים שהיה בהם שימוש אחרונה לבין אלו שלא.

כאשר מתרחש fault page, מערכת הפעלה בוחנת את כל הדפים ומחלקת אותם ל-4 קבוצות כמפורט בסעיף. האלגוריתם בוחר במקרה רנדומלית דף מהסיווג הקטן ביותר לפי סדר הא'ב שאינו ריק. במקרה זו האלגוריתם נוטן تعدוף לדפים שעברו שינוי שלא היה בהם צורך אחרונה על מחיקת דפים שהיו בשימוש תדיר.

יתרונות:

- א. קל להבנה ולמימוש.
- ב.יעילות מתונה במימוש.

ג. למרות שאינו אופטימלי, במקרים מסוימים הוא מספק ביצועים הולמים.

מוגבלות:

- א. הוא נשען על בחירה רנדומלית שאינה בהכרח ההחלטה הכי טובה.
- ב. חלוקה ל-4 מחלקות עלולה לפספס דפוס שימוש של דפים מסוימים.

למה עדיף לזרוק את B על פני C?

כי אנחנו פשוט פחות משתמשים בו. בגלל זה העלות של להביא אותו שוב בעקבות שינוי בדף תהיה יותר יקרה ולכן כבר להוריד אותו לזכור.

למה חשוב לאותחל את בית ה-R?

כדי לאפשר למערכת הפעלה להחזיק מידע עדכני.

הסדר לפי הא"ב מעיד על עד כמה יקר להוריד דף לזכרון.

הערות:

א. ניתן לסמלץ את השימוש ב-R וב-M בעזרת המנגנוןים של page fault ופסיקות שעון.

ב. הלכה למעשה האלגוריתם מעדיף את העולות של עדכון דף שבו שינויים בדיסק.

LRU - Least Recently Used replacement algorithm

- Temporal locality of reference
 - recently used pages have high probability of being referenced again
- time-expensive
 - total order of references must be managed

Bit-table implementation:

- when page i is referenced
 - set all bits of row i to 1
 - set all bits of column i to 0
- evict page with lowest binary value

	0	1	2	3
0	0	0	0	0
1	0	0	0	0
2	0	0	0	0
3	0	0	0	0

	0	1	2	3
0	0	1	1	1
1	0	0	0	0
2	0	0	0	0
3	0	0	0	0

Reference string is: 0,1,2,3,2,1,0,3

LRU - Least Recently Used Replacement Algorithm

נקודות חשובות:

א. העיקרון של Temporal locality of reference: דפים שהיו בשימוש לאחרונה יש סיכוי לכך他们会 להיות עם R-bit דולך.

ב. זמן הריצה הוא יקר כי היא דורשת ניהול של סדר הגישות.

ג. יש צורך במימוש של טבלת ביטים.

ד. כאשר יש התיחסות לדף ה-0 סמן את כל העמודה ה-0-ב-1 ואת כל השורה ה-0-ב-0. הדף עם הערך הבינארי לכך נמוך יורד מהראמם.

האלגוריתם הוכח כאלגוריתם קירוב 2. הוא לא בשימוש בדרך כלל אך כן משתמש בReLUוניות שלו. אנחנו נראה ניסיון מימוש של זה ברמת חומרה, בשביל שהוא יותר מהיר. רעיון זה לא הצליח בגלל העולות של זמן הריצה.

קירוביים:

קשה למשם בצורה מדוייקת את האלגוריתם זה ולכן ישנו מספר שיטות לתת קירוב טוב לתוצאות של האלגוריתם תוך כדי הפחתת overhead החישובי. אחד מהם הוא שימוש באלגוריתם שעון (שראינו אותו כבר).

יתרונות:

א. יעיל בהורדת מספר page fault-h-

ב. מנצל את התכונה של temporal locality.

ג. שומר על דפים בשימוש תדיר בראם ומחית את הצורך בפעולות IO בדיסק.

חסרונות:

א. עלות חישוב גבוהה. אם טכניקות הקירוב לא נוטנות את אותו הקירוב כמו האלגוריתם המדויק.

ב. הביצועים של האלגוריתם מושפעים בצורה ישירה מסדרה של ה-page reference. במקרים מסוימים הוא אפילו יכול להתבצע בצורה לא אופטימלית.

דוגמה:

יש לנו referenced string - זאת דוגמת הריצה שלנו (סדר בקשות הגישה לדפים).

יש לנו טבלה ברמת חומרה - עמודה לדף וירטואלי

בראע שיש בקשה לדף 0 שמיים אחדות על השורות ואז אפסים על העמודות מהנקודה של 0,0

לאחר מכן יש בקשה לדף 1. הולכים ל-1,1 וחוזרים על התהילה.

LRU - Least Recently Used replacement algorithm

- Temporal locality of reference
 - recently used pages have high probability of being referenced again
- time-expensive
 - total order of references must be managed**

Bit-table implementation:

- when page i is referenced
 - set all bits of row i to 1
 - set all bits of column i to 0
- evict page with lowest binary value

	0	1	2	3		0	1	2	3		0	1	2	3
0	0	0	0	0	0	0	1	1	1	0	0	1	1	
1	0	0	0	0	1	0	0	0	0	1	0	0	1	
2	0	0	0	0	2	0	0	0	0	2	0	0	0	
3	0	0	0	0	3	0	0	0	0	3	0	0	0	

Reference string is: 0,1,2,3,2,1,0,3

LRU - Least Recently Used replacement algorithm

- Temporal locality of reference
 - recently used pages have high probability of being referenced again
- time-expensive
 - total order of references must be managed

Bit-table implementation:

- when page i is referenced
 - set all bits of row i to 1
 - set all bits of column i to 0
- evict page with lowest binary value

	0	1	2	3
0	0	0	0	0
1	0	0	0	0
2	0	0	0	0
3	0	0	0	0

	0	1	2	3
0	0	1	1	1
1	0	0	0	0
2	0	0	0	0
3	0	0	0	0

	0	1	2	3
0	0	0	1	1
1	1	0	1	1
2	2	0	0	0
3	2	0	0	0

	0	1	2	3
0	0	0	0	1
1	1	0	0	1
2	2	1	1	0
3	2	0	0	0

	0	1	2	3
0	0	0	0	0
1	0	0	0	0
2	0	0	0	0
3	0	0	0	0

Reference string is: 0,1,2,3,2,1,0,3

LRU - Least Recently Used replacement algorithm

- Temporal locality of reference
 - recently used pages have high probability of being referenced again
- time-expensive
 - total order of references must be managed

Bit-table implementation:

- when page i is referenced
 - set all bits of row i to 1
 - set all bits of column i to 0
- evict page with lowest binary value

	0	1	2	3
0	0	0	0	0
1	0	0	0	0
2	0	0	0	0
3	0	0	0	0

	0	1	2	3
0	0	1	1	1
1	0	0	0	0
2	0	0	0	0
3	0	0	0	0

	0	1	2	3
0	0	0	1	1
1	1	0	0	1
2	2	0	0	0
3	2	0	0	0

	0	1	2	3
0	0	0	0	1
1	1	0	0	0
2	2	1	1	0
3	2	0	0	0

	0	1	2	3
0	0	0	0	0
1	0	0	0	0
2	0	0	0	0
3	0	0	0	0

Reference string is: 0,1,2,3,2,1,0,3

LRU - Least Recently Used replacement algorithm

- Temporal locality of reference
 - recently used pages have high probability of being referenced again
- time-expensive
 - total order of references must be managed

Bit-table implementation:

- when page i is referenced
 - set all bits of row i to 1
 - set all bits of column i to 0
- evict page with lowest binary value

	0	1	2	3
0	0	0	0	0
1	0	0	0	0
2	0	0	0	0
3	0	0	0	0

	0	1	2	3
0	0	1	1	1
1	0	0	0	0
2	0	0	0	0
3	0	0	0	0

	0	1	2	3
0	0	0	1	1
1	1	0	1	1
2	0	0	0	0
3	0	0	0	0

	0	1	2	3
0	0	0	0	0
1	1	0	0	0
2	1	1	0	0
3	1	1	1	0

Reference string is: 0,1,2,3,2,1,0,3

LRU - Least Recently Used replacement algorithm

- Temporal locality of reference
 - recently used pages have high probability of being referenced again
- time-expensive
 - total order of references must be managed

Bit-table implementation:

- when page i is referenced
 - set all bits of row i to 1
 - set all bits of column i to 0
- evict page with lowest binary value

	0	1	2	3		0	1	2	3		0	1	2	3		0	1	2	3		0	1	2	3				
0	0	0	0	0	1	0	1	1	1	2	0	0	0	0	3	0	0	0	0	0	0	0	0	1	1	0	0	
1	0	0	0	0	2	0	0	0	0	3	0	0	0	0	0	0	0	1	1	1	0	1	2	1	1	0		
2	0	0	0	0	3	0	0	0	0	0	0	0	1	1	1	1	0	1	2	1	1	0	3	1	1	1		
3	0	0	0	0	0	0	1	1	1	1	0	0	0	0	2	1	1	0	1	3	0	0	0	0	0	0	0	0

	0	1	2	3		0	1	2	3		0	1	2	3		0	1	2	3		0	1	2	3				
0	0	0	0	0	1	1	0	0	0	2	1	0	1	1	3	1	1	0	0	0	0	0	0	1	1	0	0	
1	1	0	0	0	2	1	1	0	1	3	1	1	0	0	0	1	0	0	1	1	1	0	0	2	1	1	0	
2	1	1	0	1	3	1	1	0	0	0	1	0	0	0	1	0	1	1	1	2	1	1	0	3	0	0	0	
3	1	1	0	0	0	0	1	1	1	1	0	0	1	1	2	0	0	0	0	3	0	0	0	0	0	0	0	0

Reference string is: 0,1,2,3,2,1,0,3

LRU - Least Recently Used replacement algorithm

- Temporal locality of reference
 - recently used pages have high probability of being referenced again
- time-expensive
 - total order of references must be managed

Bit-table implementation:

- when page i is referenced
 - set all bits of row i to 1
 - set all bits of column i to 0
- evict page with lowest binary value

	0	1	2	3		0	1	2	3		0	1	2	3		0	1	2	3		0	1	2	3				
0	0	0	0	0	1	0	1	1	1	2	0	0	0	0	3	0	0	0	0	0	0	0	0	1	1	0	0	
1	0	0	0	0	2	0	0	0	0	3	0	0	0	0	0	0	0	1	1	1	0	1	2	1	1	0		
2	0	0	0	0	3	0	0	0	0	0	0	0	0	0	1	0	1	1	1	2	1	1	0	3	0	0	0	
3	0	0	0	0	0	0	1	1	1	1	0	0	1	1	2	0	0	0	0	3	0	0	0	0	0	0	0	0

	0	1	2	3		0	1	2	3		0	1	2	3		0	1	2	3		0	1	2	3				
0	0	0	0	0	1	1	0	0	0	2	1	0	1	1	3	1	1	0	0	0	0	0	0	1	1	0	0	
1	1	0	0	0	2	1	1	0	1	3	1	1	0	0	0	1	0	0	1	1	1	0	0	2	1	1	0	
2	1	1	0	1	3	1	0	0	0	0	1	0	1	1	1	0	0	1	1	2	1	1	0	3	0	0	0	
3	1	1	0	0	0	0	1	1	1	1	0	0	1	1	2	0	0	0	0	3	0	0	0	0	0	0	0	0

Reference string is: 0,1,2,3,2,1,0,3

LRU - Least Recently Used replacement algorithm

- Temporal locality of reference
 - recently used pages have high probability of being referenced again
- time-expensive
 - total order of references must be managed

Bit-table implementation:

- when page i is referenced
 - set all bits of row i to 1
 - set all bits of column i to 0
- evict page with lowest binary value

Why does this work ?

Bit-table guarantees LRU since a new referenced page gets line with maximum value and does not change relative previous order of other pages.

	0	1	2	3		0	1	2	3		0	1	2	3		0	1	2	3		0	1	2	3				
0	0	0	0	0	1	0	1	1	1	2	0	0	1	1	3	0	0	0	0	0	0	0	0	1	0	0	0	
1	0	0	0	0	1	0	0	0	0	2	0	0	0	0	3	0	0	0	0	0	0	0	1	1	0	0	0	
2	0	0	0	0	1	0	0	0	0	2	0	0	0	0	3	0	0	0	0	0	0	0	0	2	1	1	0	0
3	0	0	0	0	1	0	0	0	0	2	0	0	0	0	3	0	0	0	0	0	0	0	0	3	1	1	1	0

	0	1	2	3		0	1	2	3		0	1	2	3		0	1	2	3		0	1	2	3				
0	0	0	0	0	1	0	0	0	0	2	0	0	0	0	3	1	0	0	0	0	0	1	0	1	0	0	0	
1	1	0	0	0	1	1	0	1	1	2	1	0	0	1	3	1	0	0	0	0	0	0	1	2	0	0	0	
2	1	1	0	1	1	1	0	0	1	2	0	0	0	1	3	0	0	0	0	0	0	0	0	3	1	1	1	0
3	1	1	0	0	1	1	0	0	0	2	0	0	0	0	3	0	0	0	0	0	0	0	0	3	1	1	1	0

Reference string is: 0,1,2,3,2,1,0,3

Why Bit-table implements LRU ?

Claim 1 : The diagonal is always composed of 0's.

self-read

Claim 2 : Right after page i is referenced, matrix row i has the maximum binary value (all other rows have at least one more 0 in addition to that in the i'th column)

Claim 3 :

For all distinct i, j, k, a reference to page k does not change the order between matrix lines i, j.

Proof

Assume WLOG that page j is referenced after page i. Immediately after page j is referenced, the j'th row has 1 in all entries except for entry j, whereas, from Claim 1, row i has 0 in both entries i and j (and possibly in additional entries), thus row i has a smaller value.

Any subsequent access to a page k (different from both i and j) will write 0 to entry k in both lines i and j but will not change any other entries of these lines and will therefore not change their order.



A new referenced page gets line with maximum value and does not change previous order, so a simple induction proof works.

NFU (Not Frequently Used) Algorithm

- ❑ Associate a counter with each page
- ❑ At each clock tick
 - add R-bits to counter
 - reset R-bits
- ❑ For replacement, choose page with *lowest counter*

NFU misses temporal locality – there may be a page which was intensively used in the past, and thus its counter is high. But now this page is irrelevant, and we would like to remove it from RAM.

This may be fixed by giving less weight for past references.

Example: 6 pages in RAM

		clock tick 1	clock tick 2	clock tick 3	clock tick 4
R-bits array		0 0 0 0 0 0	1 0 1 0 1 1	1 1 0 0 1 0	1 0 0 0 1 0
	5	0	1	1	2
	4	0	1	2	2
pages' counters	3	0	0	1	1
	2	0	1	1	1
	1	0	0	2	2
	0	0	1	3	4

NFU (Not Frequently Used) Algorithm

.frequently, recently, recently: נשים לב כי עד כאן ישנו רעיון שחוור על עצמו בכל האלגוריתמים שראינו:

אלגוריתם ה-NFU משייר קאונטר לכל דף בזיכרון כדי לעקוב אחר תדריות הפניות אליו. בכל טקטוק שעון האלגוריתם מבצע פעולות מסוימות כדי לעדכן את הקאונטר:

בכל טקטוק שעון, מוסיפים את ה-Rbit של כל תהליך לקאונטר שלו ולאחר מכן מאפסים אותו.

בעזרת פעולות אלו ניתן לנצל קירוב מסוים של האופן בו היה גישות לכל דף.

ברגע שמתהgesch page fault, האלגוריתם בוחר את הדף עם הקאונטר הכי נמוך, מחשב אותו כדף שהיה בו הכי פחות שימוש ומחליף אותו. במקרים אחרים, ככל שדף יהיה יותר בשימוש כך הקאונטר שלו יידל יותר.

כלל הבחירה: ברגע שיש צורך בהחלפה בחר את הדף עם ערך הקאונטר הנמוך ביותר.

דוגמא:

בהתינתן בדוגמה שבşekף זאת תהיה ריצת האלגוריתם:

Clock Tick 1:

R-bits array: 0 0 0 0 0

Counters: 5 4 3 2 1 0

Clock Tick 2:

R-bits array: 1 0 1 0 1 0

Counters: 5 4 3 2 1 0

Clock Tick 3:

R-bits array: 1 1 0 0 1 0

Counters: 5 4 3 2 1 0

Clock Tick 4:

R-bits array: 1 0 1 1 0 1

Counters: 5 4 3 2 1 0

חסרונות:

א. פספוס של עיקרון-temporal locality.

ב. דף שהיה בשימוש תDIR בעבר עדין עלול להיות בעל ערך קאנטיר גבוה, למרות שכרגע הוא לא רלוונטי.

כלומר בכל מקרה יתכן מצב שהאלגוריתם יוריד מהראמ דפים שימושיים.

זה טריד אוף די טוב, כיון שאין את המורכבות של LRU או הדיק שלו אך אין בחירה סתמית כמו ב-FIFO.

שאלה פתוחה: מהו ערך ה-page counter של page שנכנס הראג?

Aging algorithm – approximation of LRU gives less weight for past references

- Associate a counter with each page
- At each clock tick
 - shift-right all counters one bit
 - add R-bits to be counters' MSB
 - reset R-bits
- For replacement, choose page with *lowest counter*.

Note that several pages may have same counter values, which does not happen in LRU.

Example: 6 pages in RAM

		clock tick 1	clock tick 2	clock tick 3	clock tick 4	
R-bits array	5	0 0 0 0 0 0	1 0 1 0 1 1	1 1 0 0 1 0	1 1 0 1 0 1	1 0 0 0 1 0
pages' counters	5	0 0 0 0 0 0	1 0 0 0 0 0	0 1 0 0 0 0	1 0 1 0 0 0	0 1 0 1 0 0
	4	0 0 0 0 0 0	1 0 0 0 0 0	1 1 0 0 0 0	0 1 1 0 0 0	1 0 1 1 0 0
	3	0 0 0 0 0 0	0 0 0 0 0 0	0 0 0 0 0 0	1 0 0 0 0 0	0 1 0 0 0 0
	2	0 0 0 0 0 0	1 0 0 0 0 0	0 1 0 0 0 0	0 0 1 0 0 0	0 0 0 1 0 0
	1	0 0 0 0 0 0	0 0 0 0 0 0	1 0 0 0 0 0	1 1 0 0 0 0	0 1 1 0 0 0
	0	0 0 0 0 0 0	1 0 0 0 0 0	1 0 0 0 0 0	1 1 0 0 0 0	1 1 1 0 0 0

Aging Algorithm - Approximation of LRU

זהו קירוב ל-LRU אשר נותן משקל לפחות לפניות קודומות לדפים וויתר מכון לסמלי את UU בתוכנה.

לכל דף בזיכרון יש קאנטיר ייחודי.

בכל טקטוק שען:

מתבצעת פעולה זהזה ימינה בביט 1 של כל הקאנטירים.

ביט ה-R מתווסף ל-MSB של הקאנטרים.

הביט ה-R מאותחל מחדש 0.

ברגע שיש **Page Fault**, האלגוריתם בוחר את הדף עם ערך הקאנטיר היכי נמוך. מבחינת האלגוריתם זהו הדף שהיה בו היכי פשוט שימוש.

מה שמאגייע - נופל ולכן חלק מההיסטוריה נופלת.

יש כאן משהו שחרס:

ב-LRU רק אחד יכול להיות בראש הרשימה. יש כאן اي ודות. מה עדיף? 0 או 1 או 4? מי מהם יותר או פחות אטרקטיבי? זה טריידאוף שעדיין טוב לנו כי אפשר למשתמש את האלגוריתם הזה. הרעיון הוא שהאלגוריתם הזה מביא אහיכות להבחין בין גישות בזמןניים יישנים יותר באינטראול שעון כתוצאה מכמונות מוגבלת של עדכון ערך בית ה-R.

**Would Pi get less page
faults if Pi gets more
page frames ?**

NO. Let's see why.

הציפייה שלנו היא כאשר אנחנו מוסיפים עוד ראם - התוכניות יהיו מהירות יותר. זה לא טרוייאלי וזה מתקיים רק בתנאים מסוימים.

Belady's anomaly

algorithm may cause MORE page faults when given MORE page frames

Example: FIFO with reference string 0,1,2,3,0,1,4,0,1,2,3,4

	0	1	2	3	0	1	4	0	1	2	3	4
youngest frame	0	1	2	3	0	1	4	4	4	2	3	3
	0	1	2	3	0	1	1	1	1	4	2	2
oldest frame		0	1	2	3	0	0	0	1	4	4	
	F	F	F	F	F	F			F	F		9 page faults

	0	1	2	3	0	1	4	0	1	2	3	4
youngest frame	0	1	2	3	3	3	4	0	1	2	3	4
	0	1	2	2	2	3	4	0	1	2	3	
oldest frame		0	1	1	1	2	3	4	0	1	2	
	F	F	F	F		F	F	F	F	F	F	10 page faults!



Second-Chance algorithm also suffers from Belady's anomaly.
But not LRU!
Why?

To understand this, we should define stack model.

Formal modeling of page replacement algorithms

LRU example

- Reference string – sequence of page accesses made by process
- m – number of frames in RAM
- r – some prefix of Reference string
- M(m, r) – set of pages in RAM after first r requests

Page replacement algorithm is **stack algorithm** if $M(m, r) \subseteq M(m+1, r)$ for every Reference string, m and r.

Page replacement algorithm with Stack property do not suffer from Belady's anomaly.

Reference string:	0	2	1	3	5	4	6	3	7	4	7	3	3	5	5	3	1	1	1	7	1	3	4	1
m	{	0	2	1	3	5	4	6	3	7	4	7	7	3	3	5	3	3	1	7	1	3	4	
m+1	{	0	2	1	3	5	4	6	3	3	4	4	7	7	7	5	5	5	3	3	7	1	3	
		0	2	1	3	5	4	6	6	6	6	4	4	4	7	7	7	5	5	5	7	7		
		0	2	1	1	5	5	5	5	6	6	6	4	4	4	4	4	4	4	5	5			
		0	2	2	1	1	1	1	1	1	1	1	6	6	6	6	6	6	6	6	6	6		
		0	0	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2		
		0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0		

LRU orders the pages according to their usage.

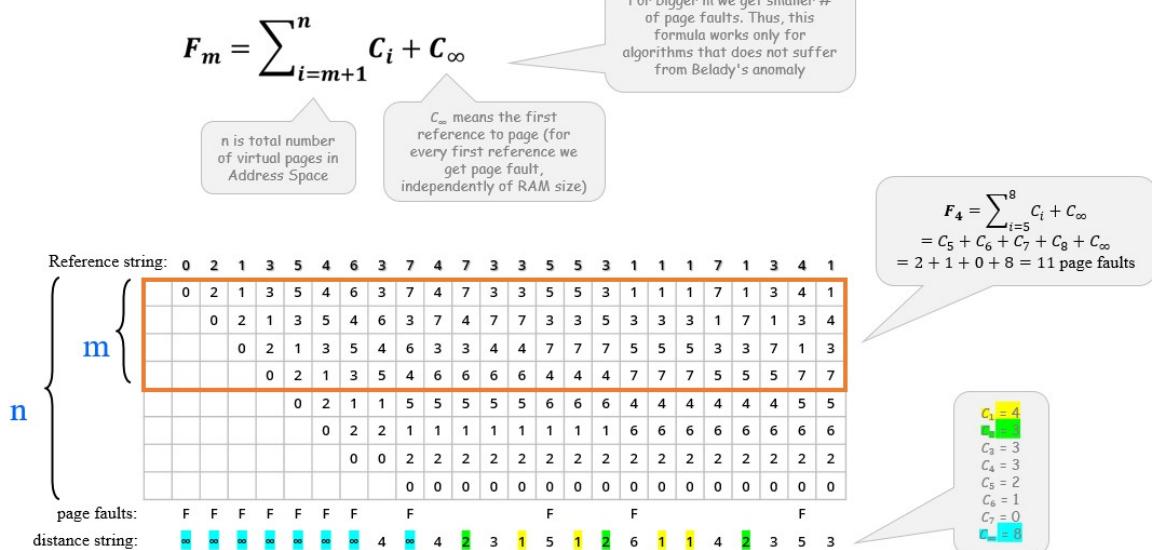
For ever choice of RAM size, say RAM with m frames, the first m pages in this order would be in RAM.
Thus, for Reference string, for every RAM size m, and for every prefix of reference string r holds: $M(m, r) \subseteq M(m+1, r)$.

If all the pages that were in RAM of size m presents also in RAM of size m+1, it is impossible to get more page faults.

Formal modeling of page replacement algorithms

Distance String - distance of referenced page from the top of stack

- C_i – number of times i occurs in distance string
- F_m – number of page faults for m frames



תופעת בלדי

תופעת בלדי היא תופעה בה ההגדלה של מספר המסגרות בזיכרון (page frames) עלולה לגרום להגברת במספר החטאות (page faults).

לדוגמה, אם השתמש באლגוריתם FIFO עם סדרת הפניות לדפים 0,1,2,3,0,1,4,0,1,2,3,4 נקבל 9 חטאות. אם נוסיף מסגרת זיכרון, נקבל 10 חטאות. זו היא תופעת בלדי.

אלגוריתם Second-Chance גם סובל מתופעת בלדי, אך לא כר אלגוריתם LRU. למה? כדי להבין זאת, علينا להגדיר את מודל המחסנית.

נגיד:

- מספר המסגרות בזיכרון m
- סדרה של סדרת הפניות r
- ביקשות i קבוצת הדפים בזיכרון לאחר i (M)

אלגוריתם החלפת דפים הוא אלגוריתם מחסנית אם לכל סדרת פניות r ו- i מתקיים $(r, M) \leq (r, M+1)$. לעומת זאת קבוצת הדפים M במסגרות לאחר i בבקשת i תחת קבוצה של קבוצת הדפים לאחר i בבקשת i במסגרת אחת נוספת. אלגוריתם החלפת דפים שאינו סובל מתופעת בלדי הוא אלגוריתם שמיים את התוכנה הזו.

אלגוריתם LRU מסדר את הדפים לפי שימושם. לכל בחירה של גודל זיכרון, אם יש לנו m מסגרות, ה- m הדפים הראשונים בסדר זהה יהיו בזיכרון. לכן, לכל סדרת פניות, לכל גודל זיכרון m , ולכל תת-סדרה i מתקיים $(r, M) \leq (r, M+1)$. אם כל הדפים שהיו בזיכרון של גודל m נמצאים גם בזיכרון של גודל $m+1$, אי אפשר לקבל יותר חטאות. **מכאן שאלגוריתם ה-LRU הוא אלגוריתם סטاك.**

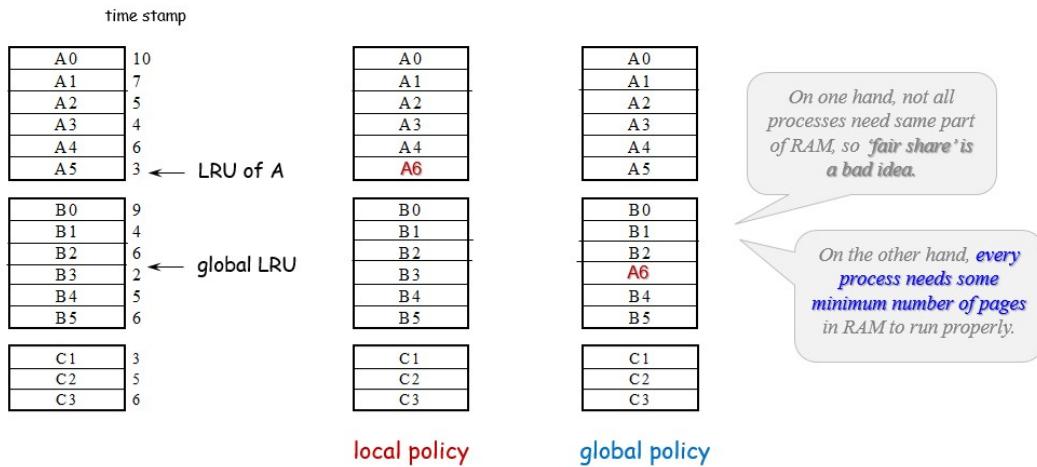
נגיד את המחרוזת המרחקית (Distance String) כمرחיק של הדף שהופנה אליו מראש המחסנית. נגיד את C_i כמספר הדפים הויירטואליים הכלולים במרחב הכתובות, ואת F_m כמספר הפעמים שהדף ה- i מופיע במחרוזת.

אלגוריתמים שאינם סובלים מהתופעה בלבד מבטיחים שהאגלת ה- m תפחית את מספר החטאות.

במילים אחרות, אפשר להסכל על כל וריאציה של אלגוריתמים כאלה כאלו האלגוריתם מבוסס מחסנית ה- c פשוט - אנחנו נקבל את אותו מספר החטאות.

Replacement algorithms in multi-process system

- **Local policy** – evicts only page of the process that causes a page fault
- **Global policy** – evicts any page among all pages in RAM



Replacement Algorithms in multi-process system

במערכות התומכות בריבוי תהליכיים ישנו 2 פוליסות הנוגעות להקצאות זיכרון בין תהליכיים מתחרים, פולישה מקומית ופולישה גלובלית.

לפי הפולישה המקומית, כאשר צריך להוריד דף, הוא יהיה של התהיליך שגרם ל-page fault. לעומת זאת, בפולישה גלובלית, ברגע שיש צורך להוריד דף מהזיכרון, ניתן להוריד דף מבין כל הדפים אשר נוכחים בראם.

ההיגיון של פולישה מקומי: אם ארגמתי ל-pagefault למה שתהליכיים אחרים ישלמו על זה.

ההיגיון של פולישה גלובלית: מערכת הפעלה רוצה לקדם את כל התהליכיים של המשתמש.

השוואה בין הפוליסות:

אלגוריתם אשר בוחן רק לפי פולישה מקומית נקרא אלגוריתם מקומי ואלגוריתם אשר בוחן לפי פולישה גלובלית נקרא אלגוריתם גלובלי.

بعد שהאלגוריתמים lokalsים אחראים להקצאות קבועות של זיכרון לכל תהיליך, אלגוריתמים גלובלים מאפשרים הקצאות דינמיות של page frames.

כמו כן אלגוריתמים גלובלים הם באופן כללי יותר ייעילים, במיוחד כאשר הגדים של ה-*working set* (ראו למטה) שונים בצורה משמעותית. אלגוריתמים lokalsים יכולים לגרום ל-"*הלקאה*" אם ה-*working set* גדל. קלומר אלגוריתם לואקי משבז זיכרון אם ה-*working set* קטן.

Replacement algorithms in multi-process system

Thrashing - high rate of page faults

- If a process **does not have minimum number of pages** to run properly, page-fault rate is very high
 - processes are busy in swapping pages in and out of RAM

In a case of thrashing, OS might mistakenly decide to increase multi-programming rate (i.e. to add more processes).

Why ?

Thrashing:

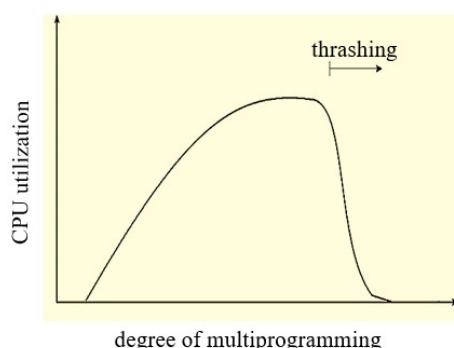
מתרחש כאשר לתהיל אין מספר מינימום של דפים הנדרשים כדי שירוץ כמו שצורך, דבר המוביל ליחס אдол יותר של page faults. במהלך ההלקה, מעבדים מוצבאים את רוב הזמן שלהם בהחלפת דפים מול הראמ, דבר המוביל לפגיעה במדד ה-*cpu utilization* ושרשתה של תוצאות בהן מערכת הפעלה בטעות מוסיפה עוד תהליכים.

Replacement algorithms in multi-process system

Thrashing - high rate of page faults

- If a process **does not have minimum number of pages** to run properly, page-fault rate is very high
 - processes are busy in swapping pages in and out of RAM
 - → **low CPU utilization**
 - → **chain reaction** – OS might think that it needs to increase the degree of multiprogramming
 - → more processes added to the system ☹

Page-fault increases IO but decreases CPU utilization. Thus OS might mistakenly think that some additional processes may be added, to increase CPU utilization.



So, how many page frames a process needs ?

Let's try to understand this
at run time.

קביעת מספר ה-page frames שתהיליך צריך

מספר זה משתנה באופן דינמי מטהיליך לטהיליך. אלגוריתמי החלפת דפים לוקאלים מקצים לכל תהיליך חלק קבוע של הזיכרון בעוד שאלגוריתמים גלובליים מקצים page frames בצורה דינמית בין תהליכיים שנייתם להריצ' אותם.

Memory management: outline

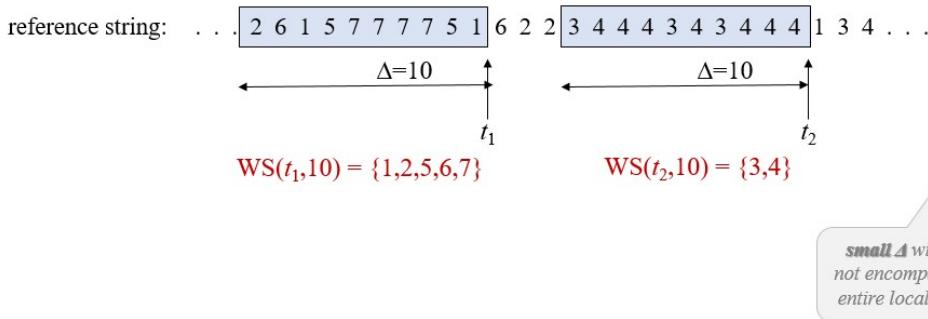
- Page replacement algorithms
 - Working set model
 - Virtual memory implementation issues

Working set model

based on Locality of Reference (temporal locality) principle

- Δ – is a fixed-size reference string window
- $WS(t, \Delta)$ – working set - is a set of pages which appear during Δ last requests of a process, up to time t

שנואה של אלגוריתם WS לאומת Aging
Aging זוכם ביחס או לא לזר ברגן
מספר כלשהו (מספר הביטים במליה) של
זמן אחורנות ועל פי זה מוחלט.
לעומת, set מתחמש בזון
ירטורי של גישה אחורנה וכן יכול לשומר
אניסומטריה שמייחת לגישה האחורנה גם
אם היא נעשאה לפני מספר רב מאוד של
clock ticks.



Working Set Page Replacement Algorithm

Working set model

עבור paging demand, דפים נטענים לזכרון רק כאשר יש צורך בהם. כאשר תהליך מתחילה לרווח אף אחד אחד מהדפים שלו נמצאים בזיכרון וכאשר התהליך יתחל לagasht לזכרון, יהיו pagefault ומערכת הפעלה תביא את הדפים הרלוונטיים. הבעיה באישה זו היא המחיר הכביד בזמן הריצה עבור מספר page faults כל פעם תחליך נטען.

לשם כך קיים מודל ה-working set - גישה בניהול זיכרון לצמצום מספר page faults ולשייפור בביבליות. לצורך המודל נגדיר: דلتא - מדובר בגודל קבוע של חלון ב-reference string.

Working Set:

ה-set working מייצג את קבוצת הדפים אשר תחליך משתמש בהם כרגע. מאוחר ולא ניתן למש את ממד זה כי שהוא בצורה עיליה, קיימים קירובים רבים. בקורס שלנו נגדיר את **the working set** כקבוצת הדפים אשר הופיעו בדلتא הבקשות האחרונות של התהליך עד הזמן t .

לדוגמה: ניקח תחליך כלשהו ונקבע דلتא להיות 10. אז ה-set working הוא בעצם 10 הגישות האחרונות של התהליך.

איך נשתמש בזה?

ניתן להשתמש בקבוצה הזאת על מנת לדעת בערך מהם הדפים האטרקטיביים לתחליך.

נקודה עדינה: צריך להיזהר לא לזרוק דפים של תחליך שנכנס הרגע.

הערה: בסופו של יומם ההרצאות הן על עקרונות מערכות הפעלה עובדות לפיהם. זה לא בהכרח אומר שככל מערכת הפעלה עובדת לפי כל עיקרון ועיקרון שאנו מכירים. כל אחת מבאייה לידי ביטוי את העקרונות שנראות לה לנכון.

לשים לב להערה באדום - היא הייתה לקריאה עצמאית.

Working set model

based on Locality of Reference (temporal locality) principle

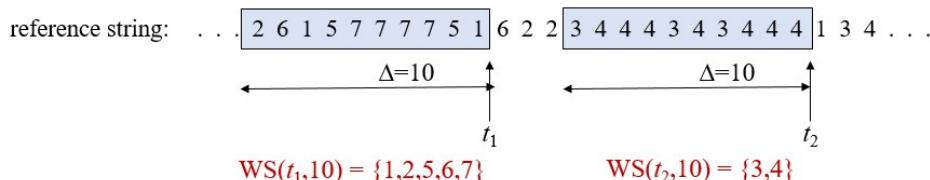
if $D_t = |WS(t, \Delta)| > m$

→ thrashing

→ some process(es) should be suspended

there is no
enough memory
to contain WS

How can we estimate
WS without doing WS
update at every
memory reference ?



עיקרון: זמן הדגימה

הרעין בגודל לפי מרינה: מנסים בכל הכוח לא לזרוק אף דף מתוך ה-set. working-set. אם יש יותר מדי' נועביר ל-. swapping area

אם ניקח דלטה קטן יותר נקבל רק גישות חממות, אם ניקח גדול יותר, נקבל גם גישות פחות אטרקטיביות. זה ערך שצורך להתאים.

תהליכיים מביאים לידי ביטוי את העיקרון של locality of reference (או temporal locality of reference) לפיו תהליכי זוכרים רק חלק קטן מהדףים שלהם בשלב מסוים בריצתם. אם כל ה-set נמצא בזיכרון, התהיליך יכול לרוץ עם מספר מינימלי של page faults, אם אין מספיק מקום להחזיק את זה, תדרות ה-page faults ירוצו לפחות. סיטואציה זו נקראת thrashing

Working set algorithm - local version

- each process P has its **clock ticks counter** C, called **P's virtual time**
- after process used CPU for additional clock tick
 - increment P's virtual time
 - update virtual timestamps for every referenced page to be current virtual time
 - clear R-bits for P's pages

The diagram illustrates the Working Set Algorithm for two processes, P1 and P2. Each process maintains a local memory table with columns for Virtual Time Stamp and R-bit.

	Virtual Time Stamp	R-bit
0	2084	1
1	2003	1
2	1980	0
3	1213	0
4	2014	1
5	2020	0
6	2032	0
7	1620	0

	Virtual Time Stamp	R-bit
0	2204	0
1	2204	0
2	1980	0
3	1213	0
4	2204	0
5	2020	0
6	2032	0
7	1620	0

משמעות:

המשמעות הבאה מומש לפי הפלישה הlokאלית:

אתחול:

כל תהליך נגדיר טבלה בגודל של כמה דפים בזיכרון.

כל דף יש את בית ה-(reference) R.

כל דף יש חתימת זמן וירטואלית - חותמת זמן לא אמיתית. יש את זה גם לכל תהליך - בכמה טיקים התהילה

השתמש (זמן האמתי יכול להיות גדול יותר).

כל שורה בטבלה מצינית את ערך ה-R-bit ואת חתימת הזמן של הדף.

תיאור הריצה:

א. לאחר שתהליך השתמש במעבד מספר ייחידות זמן:

הגדיל את הזמן הווירטואלי של התהילה, עדכן את שדה חתימת הזמן של כל דף שערך ה-R-bit דлок וכבה את בית זה

עבור כל הדפים לאחר צלול שעון שתהילה השתמש בו. ואז לכל הדפים הרלוונטיים שה-R היה דлок הוא מעדכן

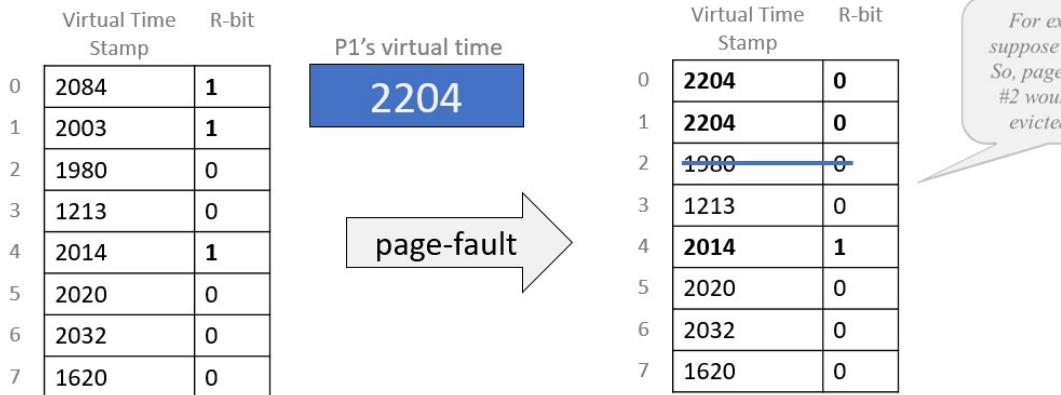
את חתימת הזמן הווירטואלית ומכבה את ה-R שליהם. (על הדרך גם מעדכן את זה של התהילה).

Working set algorithm - local version

On page-fault:

- ◻ age of page $p :=$ virtual time of $P - p$'s virtual timestamp
- ◻ for each P 's page:
 - if R-bit ==1, update virtual timestamps and clear its R-bit
 - if page's age > τ , evict this page (and break the loop)

WS is defined by last τ clock ticks rather than by last Δ memory references



ב. ברגע שהתקבלה פסיקה עקב **:page fault**
לכל דף בזיכרון של התהיליך:

נגידר את גיל הדף להיות ההפרש בין הזמן הווירטואלי של התהיליך לחתימת הזמן של הדף.
אם ה-R-bit דלאק - עדכן את חתימת הזמן של הדף וכבה את שדה זה.
אם גיל הדף עבר את הערך τ (ה-דلتא בהגדרת המודל), הוצאת דף זה מהרשימה ועוצר.

אם בסופו לא נמצאו מועמדים מתאימים, האלגוריתם בוחן בדף הבא זקן.

הסבר:

- א. ראשית נשים לב כי הטעלה מייצגת את ה-set **working set**.
- ב. שנית, מערכת הפעלה צריכה להחליט על הערך של הקבוע - המרחק עבויה הפעם האחרונה שהייתה גישה לדף הקיימת מוקסימלית (כלומר הכי זקן).
- ג. בנוסף יש לציין כי דפים מקבלים הזדמנויות שנייה רק שהם מקבלים הזדמנויות לעדכן את חתימת הזמן שלהם.
- ד. לבסוף, כפי שצוין קודם, אין דרך לשמור בצורה יעילה עילה אחר ה-set **working set**. מדובר באלגוריתם קירוב שלא מנסה להיות האופטימלי. בדוגמה בשקוף ניתן לראות כי יש דפים ישנים יותר אך לא יוצאו מהקבוצה.

Working set algorithm - local version

On page-fault:

- ❑ age of page p := virtual time of P – p 's virtual timestamp
- ❑ for each P 's page:
 - if R-bit ==1, update virtual timestamps and clear its R-bit
 - if page's age > τ , evict this page (and break the loop)

What is the difference between WS and LRU?

	Virtual Time Stamp	R-bit
0	2084	1
1	2003	1
2	1980	0
3	1213	0
4	2014	1
5	2020	0
6	2032	0
7	1620	0

P1's virtual time
2204

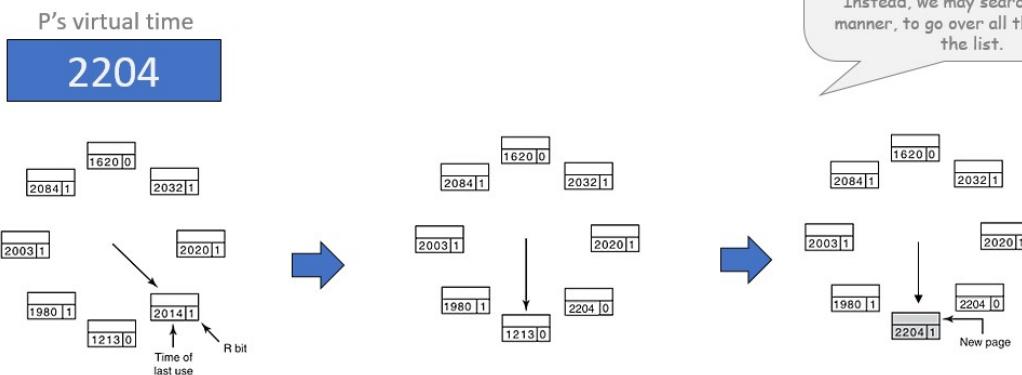
page-fault

	Virtual Time Stamp	R-bit
0	2204	0
1	2204	0
2	1980	0
3	1213	0
4	2014	1
5	2020	0
6	2032	0
7	1620	0

LRU evicts least recently used page of P , while WS may have several candidates to evict, and does not distinguish between them.
In our example, the page in frame #7 may be evicted as well.

WSClock Algorithm - local version example

like before , but using clock-like data structure



In Working set algorithm, pages with higher index have higher probability to stay in memory, since we start searching for the page to evict always from the beginning of the list.
Instead, we may search in Clock manner, to go over all the pages in the list.

The WSClock (Working Set Clock) algorithm

מדובר בגרסה מושופרת של אלגוריתם working set הבסיסי. הוא משלב את אלגוריתם השעון עם ה-*set* כדי לשפר את ביצועיו האלגוריתם.

האלגוריתם משתמש ברשימה מעגלית של פרימיטים של דפים כמבנה נתונים. לכל תא בראשימה יש את זמן השימוש האחרון (כמו באלגוריתם האחרון), בית ה-R וбит ה-(modified) M.

בכל pagefault האלגוריתם בוחן את הדף שהמחוג מצביע אליו. אם ה-Rbit דлок, זהו סימן שהדף היה בשימוש לאחרונה והוא מועמד למחיקה, אז האלגוריתם יכבה אותו ויעבור לדף הבא. תהליך זה חוזר על עצמו עד

שהאלגוריתם בוחר דף מתאים.

אם הבית כבוי והגיל שלו גדול יותר מערך קבוע תאו, האלגוריתםבודק אם הדף לא עבר שינוי ואינו נמצא ב-set. working set. אם התנאים מתקיימים, אז יש העתק ולידי של הדף בדיסק וניתן להשתמש בפריים בשביל הדף החדש. אם הדף עבר שינוי ואין עותקRALDI בדיסק, האלגוריתם מת赞同 כתיבה לדיסק תוך כדי שהוא ממשיר בסריקה.

תנאי עצירה:

אם נמצא דף מתאים - האלגוריתם עצר.

אם האלגוריתם סרק רק הרשימה פעם אחת מבלי למציא דף מתאים, ישנים 2 מקרים:

א. תזמנה כתיבה של דף אחד, וזה המחווג יסתובב עד עד שהוא ימצא דף נקי לאחר שפעולות הכתיבה הסתיימה.
ב. לא תזמנה אף כתיבה, הדבר מעיד שככל הדפים נמצאים ב-set, working set, וזה האלגוריתם יזרוק דף נקי כלשהו.

במילים אחרות, האלגוריתם משתמש בשעון על מנת לסרוק את כל הדפים של התהילה.

WSClock Algorithm - global version example

P0's virtual time	P1's virtual time	P2's virtual time
50	70	90

- window size is $\tau = 20$
- The clock hand is currently pointing to page frame 4

page-frames:	0	1	2	3	4	5	6	7	8	9	10
R-bit:	0	0	1	10	0	10	0	0	0	1	0
process ID:	0	1	0	1	2	1	0	1	1	2	2
virtual timestamp:	10	30	42	65	81	57	31	37	31	47	55
			70	70							

↑ ↑ ↑ ↑ ↑

אנו נמצאים בדף מס' 3 - האם לפנות או לא לפנות?
כל התהילים אשר נמצאים בזיכרון יש לכל אחד מהם זמן וירטואלית. נתבונן בפריים הנוכחי - ה-R דולק ולכן הוא מעדכן את הזמן הווירטואלי להיות 70 ואיז מכבה את בית ה-R.
כתוצאה לכך הדף ייחסב קטראקטיבי אלא אם התהילה בכלל לא משתמש בו באף אחד מ-20 הקלוקים ואיז הפרש הזמן יהיה יותר מיידי אдол.
עובדים על כל הדפים אשר נמצאים ברם אשר שייכים לתהילה, לא על כל הדפים של התהילה. מאחרו הקלוקים ניתנים למשוך את זה בעזרת רשיימה מקושרת.

Review of Page Replacement Algorithms

Algorithm	Comment
Optimal	Not implementable, but useful as a benchmark
NRU (Not Recently Used)	Very crude
FIFO (First-In, First-Out)	Might throw out important pages
Second chance	Big improvement over FIFO
Clock	Realistic
LRU (Least Recently Used)	Excellent, but difficult to implement exactly
NFU (Not Frequently Used)	Fairly crude approximation to LRU
Aging	Efficient algorithm that approximates LRU well
Working set	Somewhat expensive to implement
WSClock	Good efficient algorithm

סיכום

אל שמשומנים בכחול - אל שמשימוש.
رأינו את האלגוריתם האופטימלי, שלמרות שאינו ניתן למשוש, משמשCMD בבחינת כל שאר האלגוריתמים.
رأינו כי הוא בוחר את הדף שהזמן הבא שניאשים אליו הוא מקסימלי.

ה-NRU מחלק את הדפים ל-4 קבוצות לפי בית ה-R וה-M, ואז בוחר דף רנדומי מהקבוצה הקטנה ביותר. רأינו
אלטרנטיבות טובות יותר.

ה-OFO שומר רשימה מוקשת אשר עוקבת אחר סדר עליית הדפים ומסיר את הדף האחרון ברשימה. עם זאת
הוא לא לוקח בחשבון דפים חשובים עדין בשימוש - בחירה ארואה.

אלגוריתם ההזדמנות השנייה, שהוא שיפור על ה-OFO, בודק האם היה שימוש בדף לפני שנמחק. רأינו גם את
הגיסה אשר משתמש במנגנון של שעון על מנת להביא תוצאות דומות אך עם זמן ריצה מהיר יותר.

ה-LRU הוא אלגוריתם מצוין, אך דורש חומרה מיוחדת ולכן קיימים אלגוריתם קירוב עם יחס קירוב אס, ה-NFU. מצד שני רأינו את אלגוריתם ה-Aging אשר מביא קירוב טוב יותר ל-LRU ונitin למשושיעיל.

ה-working set נותן ביצועים סבירים אבל יקר במשוש שלו. בשביל זה קיים ה-WSClock אשר מביא ביצועים טובים ומשושיעיל.

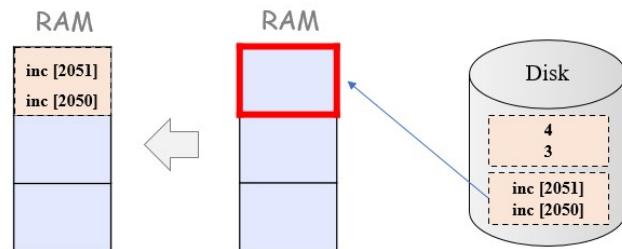
Memory management: outline

- ❑ Page replacement algorithms
- ❑ Working set model
- ❑ Virtual memory implementation issues

סוגיות בIMPLEMENTATION זיכרון וירטואלי

Locked page & busy frame

- ❑ On page fault, OS selects RAM frame to load the required page into it
- ❑ The selected frame is marked as **busy**
 - The chosen frame is marked as busy, i.e., we cannot choose it to serve some other page fault.
- ❑ The page is locked
 - The required page is locked, i.e., is exempted from being considered for replacement.
- ❑ When page arrives, update page table, mark the frame state as normal

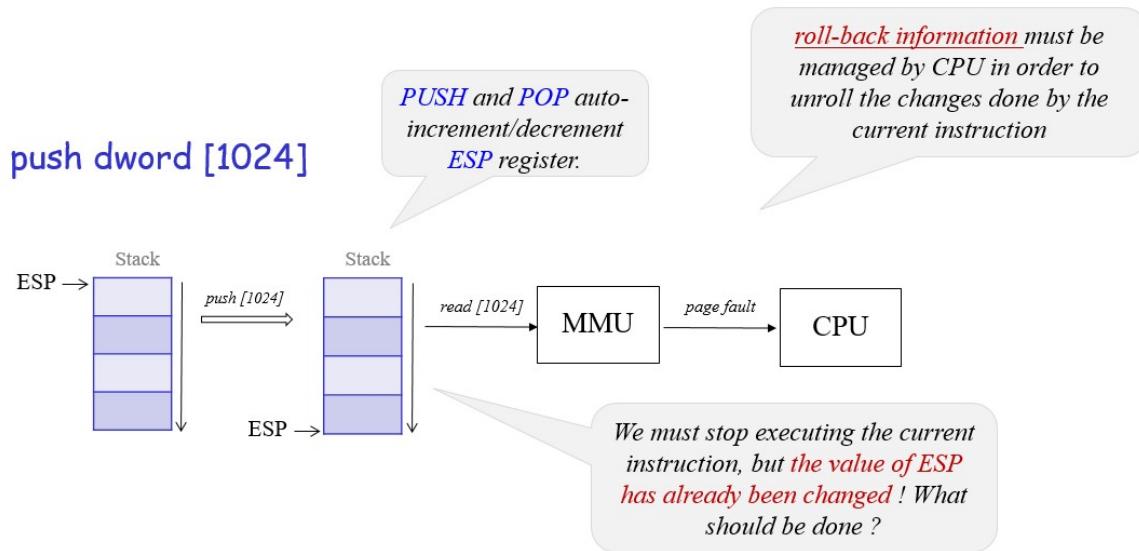


דף נעלם

כאשר תהליך שלוח בקשה ל-IO, וממתין לסיומו, הוא יכול להכנס במצב המתנה, דבר המאפשר לתהליכי אחרים ל्रוץ. אם אלגוריתם ה-paging הוא גלובלי, יש סיכוי קטן שדף שמכיל את ה-buffer so עלול לא להיות מהזיכרן. הדבר עלול להוביל לחוסר עקביות במידע בזיכרון וההתקן מעביר מידע לדף זה. לשם כך, פתרון ראשון הוא לנעלם את הדפים אשר מעורבים ב-IO בזיכרון, ובכך למנוע את הסרתם מהזיכרון. אך לא ניתן לדرس את המידע החדש המועבר מההתקן.

גישה נוספת היא על ידי ביצוע כל פעולות ה-I/O בKERNEL ולאחר מכן העתקם לדפים של המשתמש.

Interrupted Instruction Backup



המשמעות של של לנעול דף היאuai אפשר לחת אתו בחשבון בזמן שאלגוריתם החלפת דפים רץ.

במהלך הטעינה של הדף לדיסק לא נגענו בו, אבל רק עכשו הבנו מהראמ איז מה האלגוריתם יועף מהראמ.

מערכת הפעלה חייבת לפטור faultpage בזמן מהיר במיוחד. אפשר לראות את זה בלינוקס שתהליך שמתמין לדף הוא בעל עדיפות גבוהה יותר.

מקרה קצה - רדיפה אחרי דפים - יקרה בהסתברות נמוכה. אפשר למש את זה אבל זה גוזל זמן ריצה - צריך לחת בחשבון overhead כאשר מוסיפים דברים למערכת הפעלה.

נתבונן בהוראה בשקוף. זהוי פקודת דחיפה למחסנית. המילוי של המחסנית נעשת מלמעלה למיטה. לאחר מכן ניגשים ל-MMU כדי לתרגם את 1024 לכתובת האמיתית. נניח והיה **pagefault** ומערכת הפעלה תביא לו את הדף. אם נריץ ישר נקבל דף זבל בגלל הסטאק. מה שמערכת הפעלה עשויה זה לאבודת הזמן של תהליך טרם ביצוע ההוראה.

המעבד יכול לדעת על ידי שמירה של snapshot.

במעבדים מסוימים יש רגייטרים פנימיים שאיןם גלויים אשר שומרים את ה-PC ובעזרתם מערכת הפעלה יכול לבטל את ההשפעה של האירוע המתואר.

Memory access with page faults

Average RAM access time

P = probability of a page fault

MA = memory access time

PF = page-fault service time

EMA – Effective Memory Access = $(1-p) * MA + P * PF$

where

PF = page-fault interrupt service time +

Read-in page time (possibly write time of swapped-out page) +

Restart time of process

Example:

MA = 100 nanoseconds

PF = 25 milliseconds

P = 0.001

→ EMA = $(1-p)*100 + p*25,000,000$

= $(1-0.001) * 100 + 0.001 * 25,000,000 = 25,000 \text{ nano}$

אם בחרנו ב策ורה חכמה אין יותר מידי קפיצות אז נקבל הסתברות נמוכה לחטאף.pagefault.

Demand Paging vs. Pre-paging

□ Demand paging – bring a page into RAM only when it is referenced

- Less memory needed
- Potentially higher level of multi-processing
- Faster (initial) response
- Potentially more page-faults

we do not wait
for pages to be
pre-paged

If bring pages on demand, we **do not**
spend time for bringing (maybe
unnecessary) pages from disk to RAM.
Thus, only relevant pages are brought
for the process and we save a time.

□ Pre-paging – bring to RAM last working set of the process

- No page faults for the pages in working set
- Cost of bringing several pages from disk in a single read is much
cheaper than bringing these pages in separate reads

But, on the other hand, if pre-paging
was successful and we **brought**
relevant pages for the process, then
we save process time by reducing
context switches (each page fault
causes context switch, while pre-
paging of several pages may be done
together by single context switch).

Demand Paging vs. Pre-paging

בשיטת ה-Demand Paging, הבאת הדף ל-RAM מתבצעת רק כאשר יש הפניה אליו. היתרונות בשיטה זו הם הצורך בפחות זיכרון, היכולת לתמוך במגוון רחב יותר של תהליכי ותגובה מהירה בתחילת ריצת התהילר, לאחר מכן צורך להמתין להבאת הדפים מראש. אולם, היתרון זה יכול להביא גם למספר גבוה יותר של page faults.

מצד שני, בשיטת ה-Pre-paging, הדפים שהטהילר השתמש בהם בעבר מבאים מראש ל-RAM. היתרונות הם שאין תקלות הפניות דף עבור הדפים שנמצאים ברשימה, והעלות של הבאת מספר דפים בקריאה אחת היא קטנה מהעלות של הבאתם בקריאה נפרדות. בנוסף, כאשר מבאים את הדפים לפי הצורף, זמן ההמתנה מופחת

משמעותו של Pre-paging הוא מילוי ה-RAM לפני הצורך בדף. אם לא ימוצעו דפים מראש, יהיה צורך לטעינה מחדש של דפים שולטים (dirty pages) על מנת לשרת הדרישות.

בנוגע לבחירה בין השיטות בהינתן ה-set working, ב-Demand Paging, לא מבאים דף מעבר להה שתהלהיר מבקש, וזה יכול להוביל לתקלות הפניות דף. בשיטה זו, מערכת הפעלה היא זו שמחילה אילו דפים להביא. מצד שני, ב-Pre-paging, הדפים שהתחליר השתמש בהם בעבר מבאים מראש, מה שמחית את הסיכוי לתקלות הפניות דף.

Cleaning Page Frame Policy

to reduce future page-faults service time

☐ implemented by OS paging daemon thread

- periodically inspect state of RAM
- save dirty pages to disk
- if there is too few free RAM frames, run page replacement algorithm and move some frames to free list

Indeed, the selected pages are not really evicted, since their content is not overridden in RAM. They would be overridden by future page faults. Meanwhile, only their ids are moved to RAM-Frames-Free-List. If some process requests for such a page, this page would be removed from RAM-Frames-Free-List.

מדיניות ניקוי מסגרת הדפים

מדובר בפעולה המתוכנת כדי להפחית את זמן השירות של תקלות הפניות הדפים בעתיד. המדיניות מומשת על ידי תרד שמתוכנת במערכת הפעלה, הנקרא "paging daemon". כל כמה זמן התרד בודק את מצב ה-RAM: הוא שומר דפים "מלוכלים" על הדיסק, ובמקרה שיש מעט מאוד מסגרות RAM פניות, הוא מפעיל את אלגוריתם ההחלפה של הדפים ומעבר כמה מהם לרשותה הפניה.

עם זאת, הדפים הנבחרים אינם מצויים באמת מה-RAM, כיון שתוכנם אינם מוחלט. הם יוחלפו בתקלות הפניה בעתיד. בינוים, רק המזהים שלהם מועברים לרשותה הפניה. אם תחליר כלשהו מבקש את אחד מהדפים הללו, הדף יוסר מרשותה הפניה.

המבנה מאחורי מדיניות הניקוי

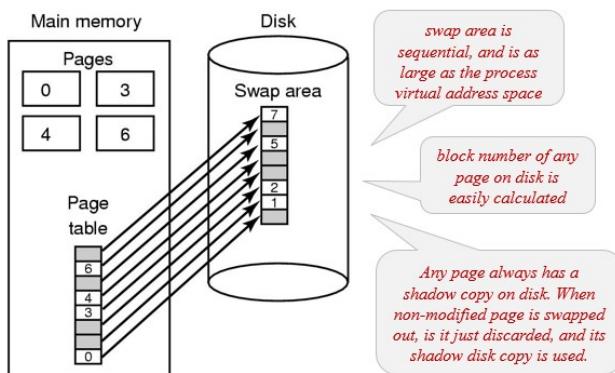
הנושא קשור לבקרת העומס ולבעיית הניקוי. עיבוד ההזדקנות מתבצע באופן הטוב ביותר כאשר יש מלאי אdoll של מסגרות דפים פניות שניית לתפוס כאשר מתרחשות תקלות הפניה. במקרה שככל מסגרת הדפים מלאה, והיא השנתנה, לפני שנית להביא דף חדש, יש לכתוב את הדף הישן לאחסן. על מנת להבטיח מלאי אдол של מסגרות דפים פניות, מערכות הפעלה מחזיקות תחליר רך, הנקרא "paging daemon", שרוב הזמן הוא במצב שינה אך מתעורר מדי פעם לבחינת מצב הזיכרון.

אחד הדריכים למשתמש את מדיניות הניוקי היא באמצעות "שעון דו-ידי" (ראה הסבר נוסף בהמשך בינויקס). היד הקדמית מופעלת על ידי daemon.paging. כאשר היא מציבעה על דף מסוילך, הדף מוחזר לאחסון לא הטעוף והיד הקדמית מתקדמת. היד האחורי משמשת להחלפת הדף, כפי שבאלגוריתם השעון הסטנדרטי.

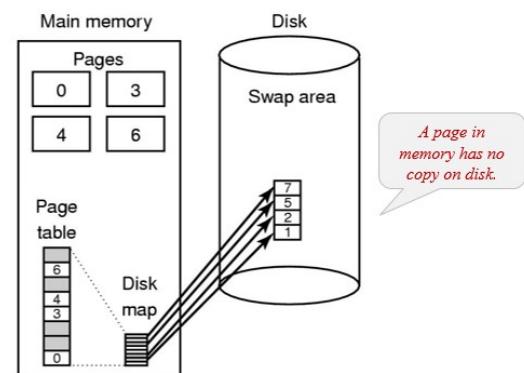
בסק הכלל, מדיניות זו יכולה להיראות מיותרת בהתאם, אך היא מאפשרת המערכת להמשיך לפעול באופן חלק במקום להמתין לתהיליך. העדפת ה-*swap daemon* היא נמנוכה, אך התוצאה היא פעולה ייעילה וביצועים גבוהים יותר.

Handling the backing store

- Static Swap area :** allocate a fixed chunk of the swap area upon a process creation
 - o problem: memory requirements of Stack and Heap change dynamically



- Dynamic Swap area :** allocate swap space for a page when it is swapped out, de-allocate when it is swapped in
 - o problem: need to keep Disk map table with swap address on Disk for each page



אזורים החלפה במערכות הפעלה

במהלך תהליך Paging, ישנו שני אסטרטגיות עיקריות לניהול אזור ה החלפה:

אזור ה החלפה סטטי (Static Swap Area)

בmethod זה, בעת יצירת תהליך, המערכת מקצה לו אזור ה החלפה בגודל קבוע. היתרון בmethod זה הוא בפשטותו וביכולת לחשב בקלות את מקום הדף באיזור ה החלפה. עם זאת, היא אינה גמישה, שכן ה-Stack וה-Heap יכולים להשתנות בגודלם לאורך הזמן.

אזור ה החלפה דינמי (Dynamic Swap Area)

בmethod זה, המערכת מקצה איזור ה החלפה לדף כאשר הוא מוסתר מהזיכרון ומשחררת אותו כאשר הוא מוחזר לזכרון. היתרון הוא יכולת להתאים את עצמה לתהליכי אופן דינמי, אך החסרונו הוא הצורך בניהול זיכרון מתקדם יותר. לכן, יש צורך במנגנון שיכל למצוא ולהחזיר את הדף מהdisk בזורה מהיריה ויעילה. לשם כך, משתמשים במאפה של הדיסק, או "Disk Map", המציבעה על המיקום הפיזי של כל דף באיזור ה החלפה.

ה-Disk Map מכילה כתובות לכל דף וירטואלי בתהליך. כאשר הדף נמצא בזיכרון הראשי, הכתובת שלו אינה פעילה. רק כאשר הדף מוסתר, הכתובת מתעדכנת כך שתצביע על המיקום הפיזי של הדף באיזור ה החלפה.

שאלה: למה לא כדאי להשתמש בטבלת האש?
תשובה: בטבלת האש היא כל'וiesel לחיפוש מהיר של מידע במבנה נתוניים, אך לא בהכרח הכל'י הנכון בהקשר זהה. זמן החיפוש בטבלת האש עשוי להיות אדול יותר, במיוחד כאשר ישנו הרבה התייחסויות. הפתרון המועדף לכך הוא הרחבת הטבלה או שימוש בפונקציית האש אחרת, אך זה יכול להוביל לבזבוז משאבים נוסף. בנוסף, השימוש בטבלת האש מחייב את המערכת לבצע פעולות נוספות של הכנסה והוצאה של דפים, שיכולה להיות יקרה בזמן ריצה.

ניהול איזור החלפה במערכות הפעלה שונות:

מערכות הפעלה רבות, כמו UNIX, משתמשות במחיצת החלפה מסוימת בדיסק. אך לעיתים, בגלל הרחבה דינמית של התהליכים, ניתן והגודל הקצוב מראש לא מספיק. במקרה זה, ניתן שיטות נוספות לניהול איזור החלפה, כגון שימוש באיזור החלפה דינמי או שיטות אופטימיזציה אחרות.

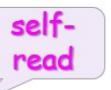
סיכום נוספים:

על אף היתרון והחסרונות של כל שיטה, חשוב להבין את התוצאות המעשיות של כל בחירה: הבחירה באיזור החלפה סטטי עשויה להוביל לבזבוז מקום אחסון, בעוד שהבחירה באיזור דינמי יכולה להגביר את זמן הגישה לדיסק.

Operating system involvement in paging - summary

□ Upon new process creation

- Determine initial size of code + data (according to exe file)
- Create a page table: allocate space, initialize
- Allocate space in swap area on disk
- Initialize swap area
- Update info about page-table and swap area in PTE



□ Upon process scheduling for execution

- Reset MMU (ptbl pointer) and flush TLB
- Select scheduled process' page table as current
- Optionally bring some of the process' pages to memory (pre-paging)

□ Upon process exit

- Release page table, pages and swap area
- Don't release shared pages if still referenced

התערבות מערכת הפעלה ב-Paging

יצירת תהליך חדש:

במהלך היצירה של תהליך חדש במערכת בה השתמשים ב-Paging, מערכת הפעלה צריכה לקבוע את הגודל ההתחלתי של הקוד והנתוניים. לאחר מכן היא יוצרת טבלת דפים, מקצתה לה מקום בזיכרון ובמצעת אתחול. המערכת מקצתה גם מקום באיזור החלפה בדיסק ומבצעת אתחול לאיזור זה. בסיום התהליך, המערכת מעדכנת את המידע הקשור לטבלת הדפים ולאיזור החלפה ב-PTE.

זמן תחילת הפעלה:

לפני הפעלת תחילת, יש לאפס את ה-MMU ולנקוט את ה-TLB. המערכת אז בוחרת בטבלת הדפים של התהליך

המתזמנ והופכת אותה לנוכחית. במקרה הצורך, המערכת יכולה להעביר את כל הדפים או חלק מהם לזכרון על מנת לצמצם את מספר השגיאות בהפעלת התהיליך.

ייצוג שגיאות דף:

כאשר מתרחשת שגיאת דף, המערכת קוראת את הרשומות מהחומרה לצורך זיהוי הכתובת הוירטואלית שארמה לשגיאה. המערכת מחשבת איזה דף נדרש ומוצאת אותו באחסון הלא נדייף. המערכת אז מוצאת מקום זמין בזיכרון, מעבירה את הדף הנדרש לתוכו ומכונת את מחשב התוכנית להפעלת הפקודה שנכשלה שוב.

יציאת תהיליך:

כאשר תהיליך מסתיים, המערכת משחררת את טבלת הדפים שלו, את הדפים ואת האחסון הלא נדייף שמכיל את הדפים. אם דפים מסוימים משותפים לתהיליכים אחרים, הם ישוחררו רק כאשר התהיליך האחרון שמשתמש בהם יסתיים.

Handling page faults - summary

- ❑ MMU sends a hardware interrupt, PC saved on stack
- ❑ Registers are saved, kernel (page-fault OS handler) is called
- ❑ Kernel discovers the virtual page that caused fault
- ❑ Kernel checks *valid* bit and verifies protection.
If illegal access – send signal to process.



Otherwise:

- Check for a free frame.
If non available, *apply a page replacement algorithm*.
- If selected frame is dirty – write it to disk
 - meanwhile run some other process - context switch
 - mark frame as busy
- When frame is clean, bring the required page from disk (process still suspended)
- When page arrives, update page table, mark the frame state as normal
- Upon process re-scheduling, re-execute faulting instruction, reload registers, continue execution

טהיליך טיפול בשגיאות דף

1. אירוע הפעלה ותגובה ראשונית:

ה-MMU שלוחת הפסקת חומרה, וריג'יסטר ה-PC מגובה במחסנית, כמו גם הרשומות ומידע אחר כדי שמערכת ההפעלה לא תפגע בו. לאחר מכן, המערכת קוראת למנהל השגיאות של הדף.

2. זיהוי הדף הבועתי:

המערכת מנסה לזהות איזה דף וירטואלי דרוש. במידה והמידע אינו זמין ברשומות החומרה, המערכת צריכה לפענча את ההוראה בתוכנה כדי לזהות את הבעיה.

3. בדיקה ואיומות:

המערכת בודקת אם הכתובת הוירטואלית תקינה ואם יש הרשות מתאימות לדף. במידה ויש בעיה, התהיליך מקבל סייגל או נהרג.

4. מניעה והחלפה:

אם הכתובת תקינה, המערכת בודקת אם יש זיכרון פנוי. אם אין, מתבצעת אלגוריתם החלפת הדף.

5. טיפול בזכרון "מלוכלך":

אם הזיכרון הנבחר הוא "מלוכלך", הדף מתוכנן להעברה לאחסון ותהליך המערכת מושהה.

6. הבאת הדף הנדרש:

כשהזיכרון נקי, המערכת מוחפשת את הכתובת בדיסק בה הדף המבוקש מאוחSEN ומתוכננת הבאה מהDISK.

7. עדכן טבלת הדפים:

לאחר הבאת הדף מהDISK, טבלת הדפים מתעדכנת והמצב של הזיכרון מתעדכן כ"גAIL".

8. החזרת ההוראה הביעיתית:

ההוראה האורתוגונאלית לבעה מוחזרת למצובה הראשוני ומחשב התוכנית מופנה אליה.

9. מתזמנת התהיליך הביעיתי:

התהיליך הביעיתי מתזמן מחדש והמערכת חוזרת לrutinen שקרה אליה.

יתרונות וחסרונות:

Virtual Memory - Advantages

- Programs may use much smaller RAM than their maximum requirements
 - Higher level of multiprogramming
- Programs can use much larger virtual memory than RAM
 - simplifies programming and enable using powerful software
- External fragmentation is eliminated
- More flexible (per page) memory protection

Virtual Memory - Disadvantages

- ❑ Special hardware (**MMU**) for address translation - some instructions may require several address translations
- ❑ **Complexity of OS**
- ❑ Overhead - a **page-fault** is an **expensive operation** in terms of both CPU and I/O overhead
- ❑ **Thrashing problem**

Memory management: outline

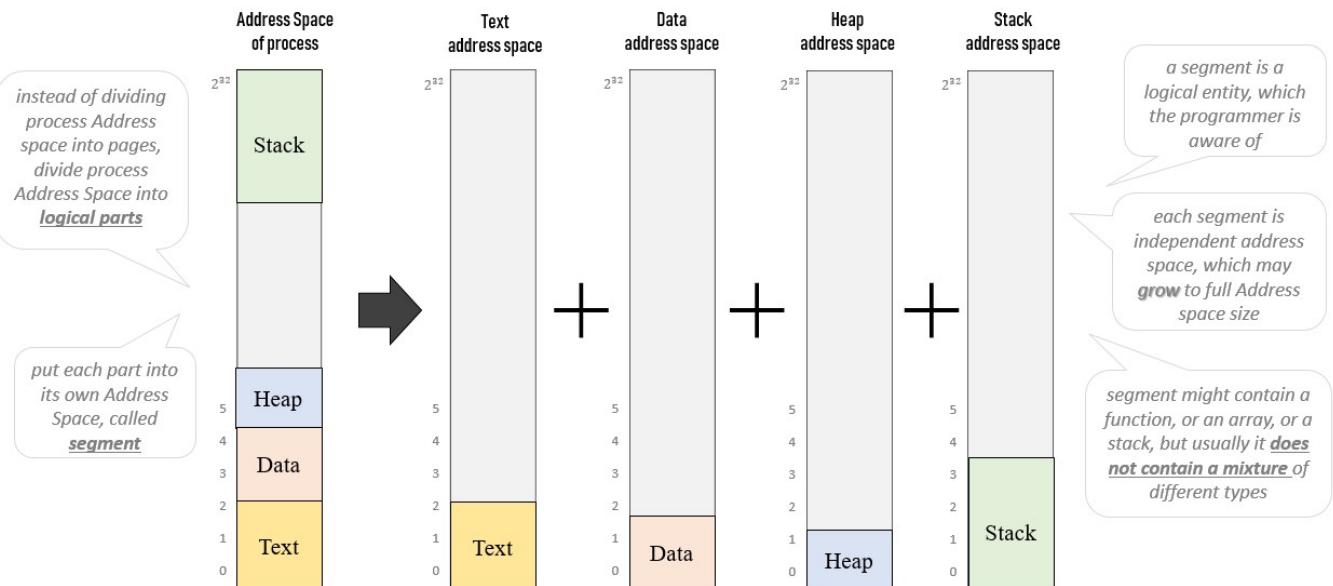
❑ Segmentation

❑ Case studies

- MULTICS
- x86 (Pentium)
- Unix
- Linux
- Windows

Segmentation

several virtual address spaces per process



Segmentation

several virtual address spaces per process

Example: suppose we use **separate segments** for

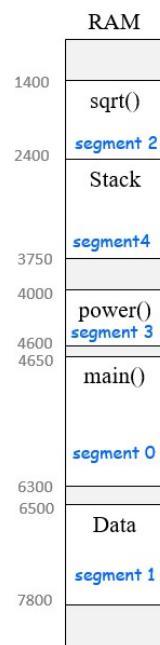
- o main() function
- o sqrt() function
- o power() function
- o Data
- o Stack

Each Segment may be divided into logical subsegments.
For example, we may divide section Text to functions, and manage a separate segment for each function.

Since (non-paged) segment are mapped sequentially into RAM, this leads to external fragmentation

segment table

	limit	base
0	1650	4650
1	1300	6500
2	1000	1400
3	600	4000
4	1350	2400



Segmentation Architecture

□ **Segment table** – maps segments to RAM

- o **base** – start physical address of segment
- o **limit** – segment size

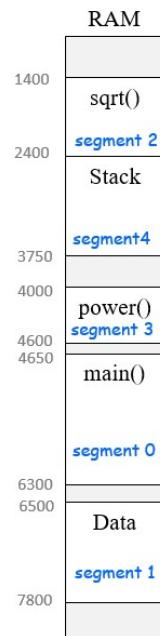
- **Segment-table base register (STBR)** points to the segment table's location in memory
- **Segment-table length register (STLR)** indicates number of segments used by a program

□ **Protection:** each segment table entry contains:

- o validation bit = 0 ⇒ illegal segment
- o read/write/execute privileges

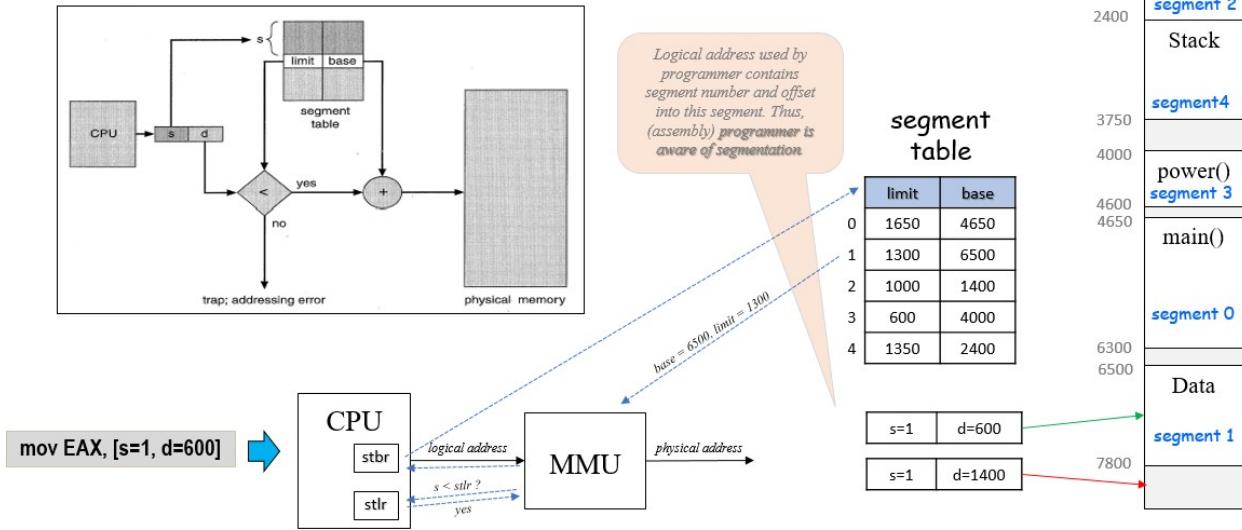
segment table

	limit	base
0	1650	4650
1	1300	6500
2	1000	1400
3	600	4000
4	1350	2400

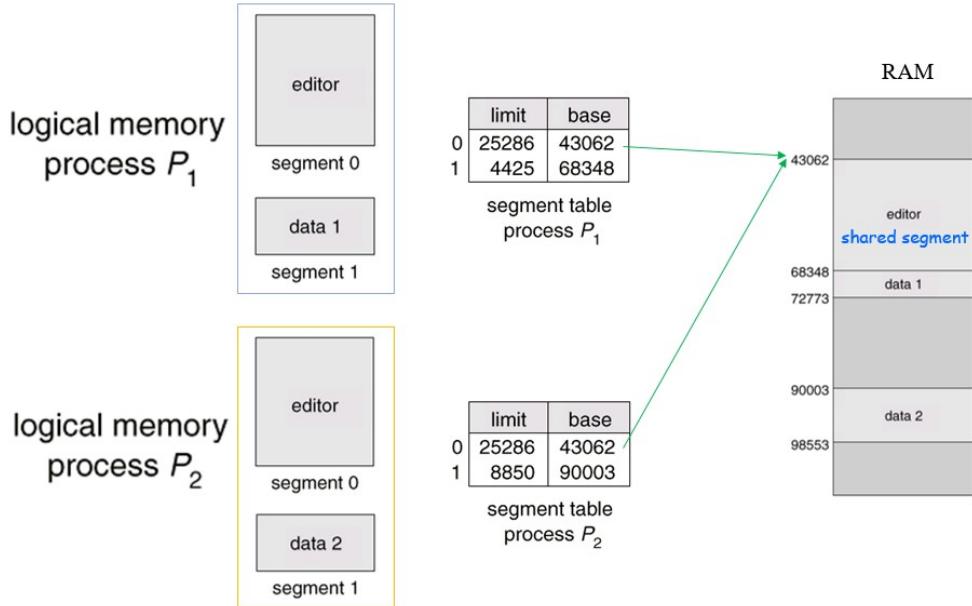


Segmentation Architecture

□ Logical address composed of the pair <segment-number, offset>



Shared segments



Segmentation vs. paging

- **Advantages:**

- each segment grows independently
- sharing segments between processes is simpler
- segments protection is easier
- linking is easier

larger overall virtual address space

- **Disadvantages:**

- segments are sequential in RAM → external fragmentation
- Full segments must be in RAM.
What if segments are very large and don't fit into RAM ?

only **changed segment** should be **recompiled**, since changes in one segment do not affect virtual addresses in other segments

Maybe we should divide segments into pages ?

Segmentation

מהו "Segmentation" ?

במערכות הפעלה, הגישה של "Segmentation" מתייחסת לחלוקת מרחב הכתובות של התהילר לSEGMENTS לוגיים. כל SEGMENT מייצג מרחב כתובות ייחודי ויכול להכיל סוגים שונים של תוכן: קוד, נתונים, ערים מהחסנית.

למה זה טוב ?

א. **אמישות ויעילות בניהול הזיכרון:** בኒואוד ל-"**paging**", SEGMENTS יכולים להיות מוגבלים או להתכווצת באופן דינמי בלחטי תליי.

ב. **תקשרות בין תהליכיים:** SEGMENTS יכולים להיות משותפים, מה שמקל על התקשרות בין תהליכיים שונים.
ג. **אופטימיזציה:** ניתן לטעון לזכרון רק חלק מהSEGMENTS כאשר הם נדרשים.

איך זה עובד ?

: **ארQUITקטורת Segmentation**

המערכת משתמשת בטבלת SEGMENTS שסמן את SEGMENTS לכתובות הפיזיות בזיכרון. עבור כל פניה לזיכרון, המעבד מבצע תהליכיים שונים לחישוב הכתובת הפיזית.

שדות בטבלת SEGMENTS:

מורכבת מהבאים, שהיא כתובות התחלת של SEGMENT, לIMIT, שזא אודל SEGMENT, ה-STBR וה-R-STR, שם רגיסטרים המצביעים למקום טבלת SEGMENTS ומספר SEGMENTS בשימוש על ידי התוכנית .
הכתובת הלוגית מורכבת ממספר SEGMENT והתיסט מתחילה SEGMENT. בנוסף ניתן לשתף SEGMENTS.

יתרונות וחסרונות :

יתרונות :

א. אמישות בניהול הזיכרון והתאמאה לצרכים הדינמיים של התהילר.

- ב. תקשורת בין תהליכיים ושיתוף סאגמנטים.
- ג. האנה יעליה יותר על המידע.
- ד. קל יותר לבצע linking לתוכניהם.
- ה. רק סאגמנטים שעברו שינוי מוחיינים בקימפול חדש - דבר המיעל את תהליך הקימפול.

חסרונות:

- א. פרוגרנמאנטייה חיונית בזיכרון - יש צורך באיחוי לראם.
- ב. הוצרך לטען סאגמנטים שלמים לזכרון, אשר יכולים להיות גדולים מדי.
- ג. סאגמנטים מלאים חייבים להיות בראם.
- ד. מרחב הכתובות הווירטואליות גדול יותר.

הערה: גודל הטבלה הוא כתלות במספר הסאגמנטים..

Memory management, part 3: outline

Segmentation

Case studies

- MULTICS (1969-2000)
- x86 (Pentium)
- Unix
- Linux
- Windows

Case Studies

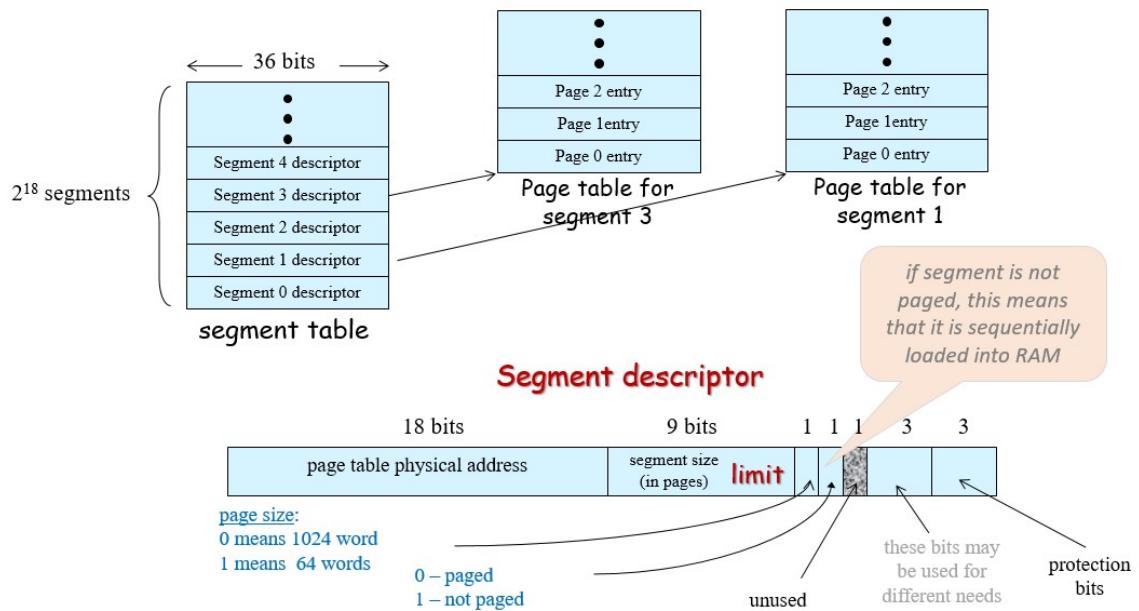
MULTICS OS

segmentation + paging is used

- Up to 2^{18} segments per process
- Segment length up to 2^{16} 36-bit words (~ 64 K words)
- Each segment has its page table
- Segment table is kept in separate segment



MULTICS data structures



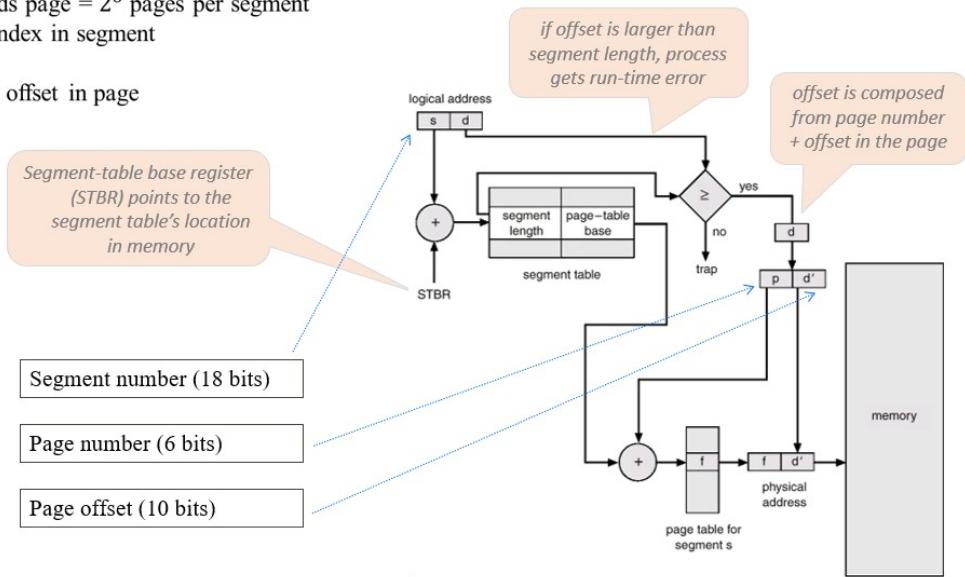
MULTICS Address Translation Scheme

64 K words segment / 1 K words page = 2^6 pages per segment

→ 6 bits are needed for page index in segment

$1024 = 2^{10}$ words page

→ 10 bits are needed for word offset in page



MULTICS

מערכת הפעלה MULTICS משלבת את השיטות של ה-segmentation וה-paging כדי ליהנות מיתרונותיהם של שתי הטכניקות.

המבנה של המערכת:

- סגמנטציה: יש למערכת יכולת לתמוך بعد 2^{18} סגמנטים לכל תהליך.
- גודל הסגמנטים: כל סגמנט מכיל עד 2^{16} מילימ של 36 ביטים (כ-64 אלף מילימ).
- ה-page table של כל סגמנט: כל סגמנט מגיע עם ה-page table הייחודי שלו.
- טבלת הסגמנטים: מוחזקת בסגמנט נפרד, מאפשרת ניהול של סגמנטים רבים וקטנים.
- ה-רגיסטר STBR: מצביע על מקום טבלת הסגמנטים בזיכרון.

אוף המרת הכתובת:

Address Translation Scheme:

הכתובת מכילה שלושה מרכיבים:

- מספר הסגמנט: 18 ביטים
- מספר הדף: 6 ביטים
- ה-offset בתוך הדף: 10 ביטים

התהליך הוא שילוב של שתי השיטות: המערכת מבצעת גישה לטבלת הסגמנטים כדי למצוא את הסגמנט הרצוי ובמקביל משתמשת בגישת ה-paging כדי למצוא את המילה הרצiosa בתוך הסגמנט.

אופטימיזציות ביצועים:

מערכת הפעלה השתמשה ב-TLB (Translation Look-aside Buffer) כמטען מהיר להמרת הכתובות. TLB שומר את הכתובות הווירטואליות הנוכחיות ואת מספר הפריים המתאים להן. בזמן בקשת גישה לכתובת, TLB היה הראשון להיבדק.

Memory management, part 3: outline

❑ Segmentation

❑ Case studies

- MULTICS

- x86 (Pentium)

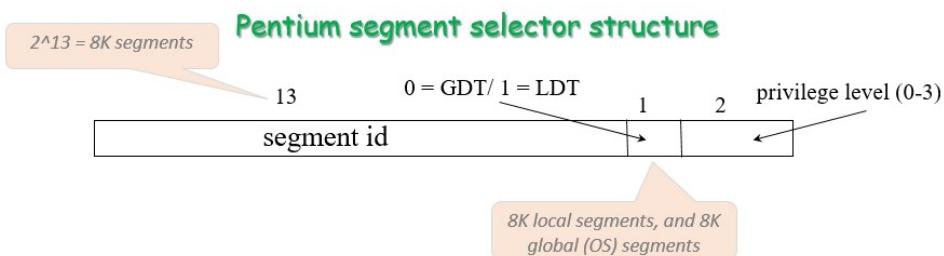
- Unix

- Linux

- Windows

Pentium (Intel): segmentation + paging

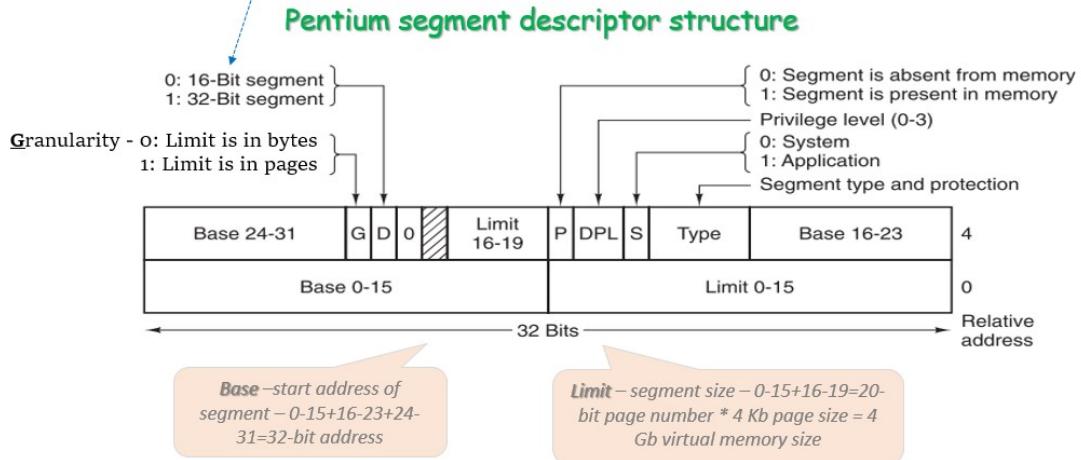
- Segmentation with or without paging is possible
- 16K segments per process, segment size up to 4G 32-bit words
- 4Kb page size
 - **GDT** – Global Descriptors Table **of kernel** (up to 8K segments)
 - **LDT** – Local Descriptors Table **per process** (up to 8K segments)
- 6 segment 16-bit hardware registers may store **segment selectors**: CS, DS, SS...



Pentium (Intel): segment descriptors

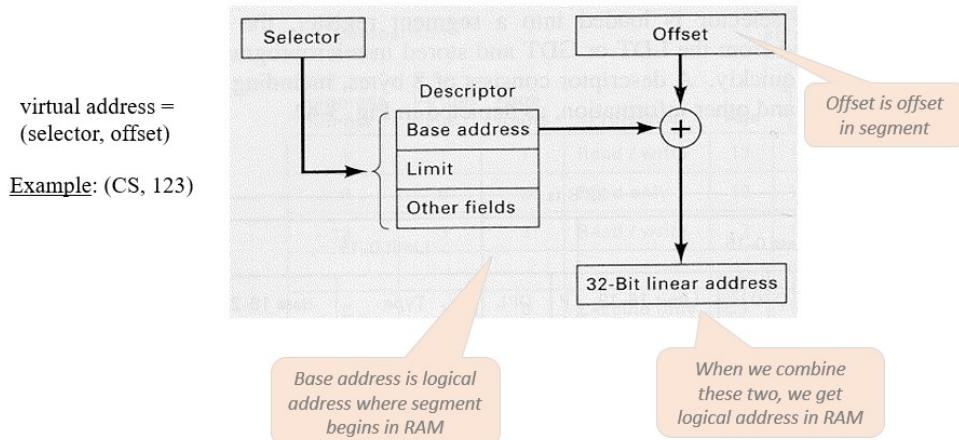
Backward compatibility requires non-linear descriptor fields partition

- When selector is loaded to a segment register, the corresponding descriptor is loaded into some (hidden) register

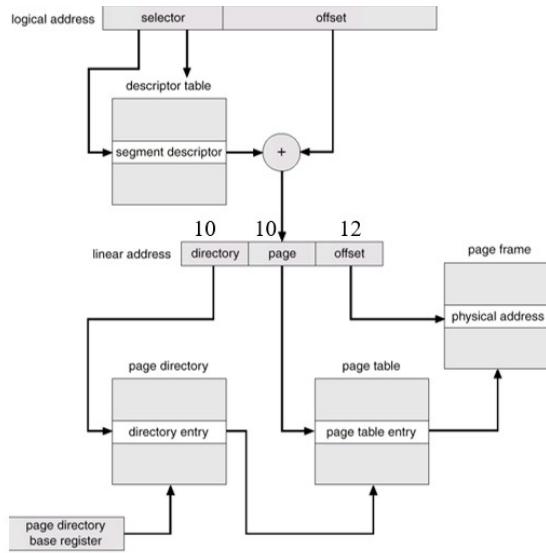


Pentium (Intel) - forming linear (i.e., non-paged) address

- Offset size is checked against *limit* field of descriptor
- Base* field of descriptor is added to *offset*



Pentium (Intel) paged address translation



Pentium (Intel)

ניהול הזיכרון בארכיטקטורת (Intel)

מעבד הפנטום של אינטל הוא מיוחד בהקשר של ניהול זיכרון. תהליך התכנון שלו היה בזמן שלא היה ברור באיזו גישה עדיף לנהל את הזיכרון: בעזרת סגמנטים או בעזרת דפים. لكن הארכיטקטורה של אינTEL תומכת בניהול זיכרון עם ניהול דפים ובלי ניהול דפים.

ניהול סגמנטים:

שתי הקונפיגורציות העיקריות הן: עם מיפוי עמודים ובלודי. בשיטה זו, קיימים 16 סגמנטים לכל תחילה, וכל סegment יכול להיות בגודל של עד 4 גיגה מיליב-ב-32 ביט. לניהול סגמנטים, הארכיטקטורה משתמשת בשלושה רגיסטרים עיקריים, אשר מהווים בוררי **סלקטור**: DS, CS ו-SS. התהליך השתמש גם בשני טבלאות: GDT למערכת ו-LDT לתהילה, כאשר האחורה משמשת לסגמנטים לוקליים והריאשטיים לסגמנטים הכלולים את קוד המערכת.

טבלאות ורגיסטרים:

לניהול הסגמנטים ולזיהוי הסגמנטים בזיכרון, נעשה שימוש ברגיסטרים כמו DS, CS ו-SS. כמו כן, מצוינות שני טבלאות חשובות: GDT המכילה את תיאורי הסגמנטים הגלובליים (עד 8K תיאורים) ו-LDT המכילה את התיאורים המקומיים לתהילה (עד 8K תיאורים).

השימוש ב-GDT מקל על הגישה לקוד במערכת הפעלה ומשפר את ביצועי הקריאה בכך שהוא חוסר קפיצות דרך הקernel.

יצירת כתובות לינאריות (כלומר כתובות ללא דפים).

כאשר בורר הסגמנט נתען אל רגיסטר סגמנט כלשהו, המזהה המתאים נתען אל רגיסטר כלשהו. רגיסטר זה מוסתר מעיני המשתמש, ככלומר לא ניתן לתוכנת אותו.

פרטים חשובים במבנה ערך בטבלת פירוש הסאגמנטים כוללים את כתובות הבסיס (Base) - כתובות ההתחלה של הסאגמנט בזיכרון, ואת האבול (Limit) - הגודל המרבי של הסאגמנט.

כאשר נדרש גישה לכתובות בסאגמנט, יש שימוש בנוסחה הבאה לתרגום הכתובות: "כתבת הזיכרון הלוגית = סלקטור * מוחול + האופסט". כתובות המוחול היא הכתובות הפיזיות בסאגמנט, והאופסט הוא המהלך מההתחלה של הסאגמנט.

תרגום כתובות לוגיות (עם דפים):

בגישה זו הכתובות הלוגיות כוללות אינדקס בטבלת הרשומות (הborer) ואופסט. דרך הרשומה בטבלה יחד עם היחסן מתאפשרת הכתובות הלינארית. בעזרת 10 הביטים הראשונים בכתובות ניתן להגיע לרשותה ב-page directory (להלן למעשה לטבלה ברמה הראשונה). בעזרת הכתובות ברשותה זו ו-10 הביטים הבאים בכתובות הלינארית, ניתן להגיע לטבלה הדפים (להלן למעשה לטבלה ברמה השנייה) ובעזרת 12 הביטים האחרונים בכתובות הלינארית והכתובות בטבלת הדפים ניתן להגיע לכתובות הפיזיות.

Memory management, part 3: outline

- ❑ Segmentation

- ❑ Case studies

- MULTICS

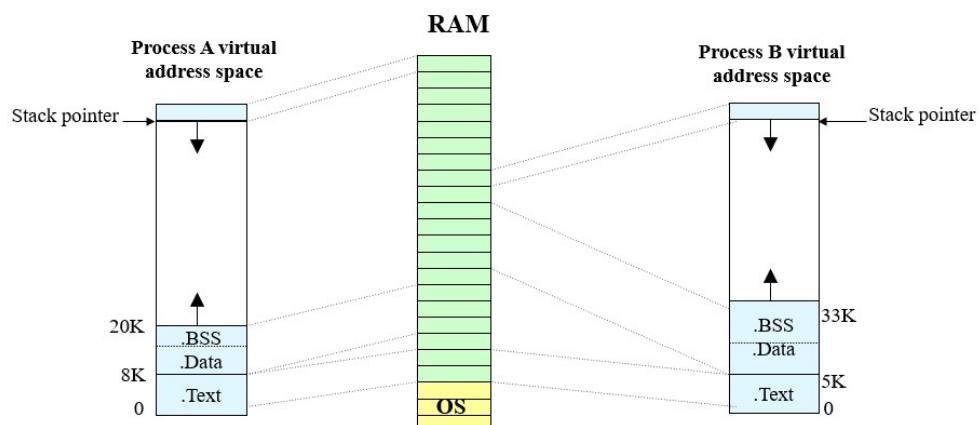
- x86 (Pentium)

- Unix

- Linux

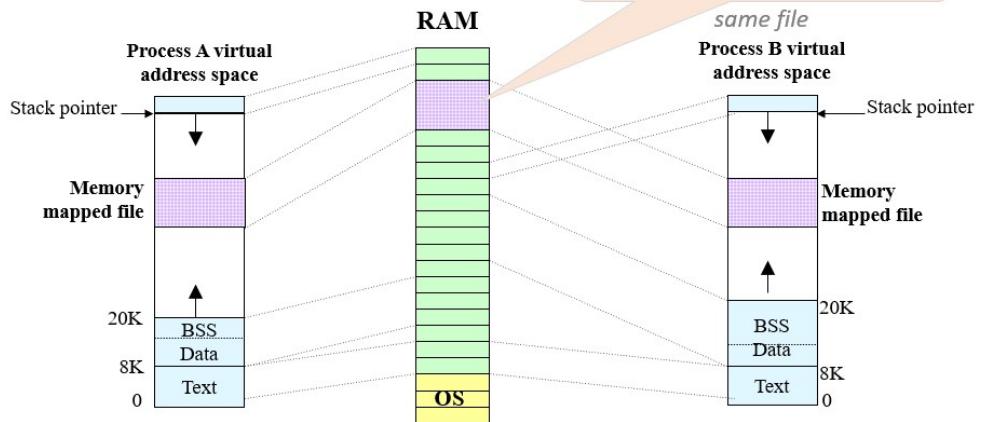
- Windows

UNIX process address space



Memory-mapped files

in this way processes may communicate one with other, since they would write to the same file

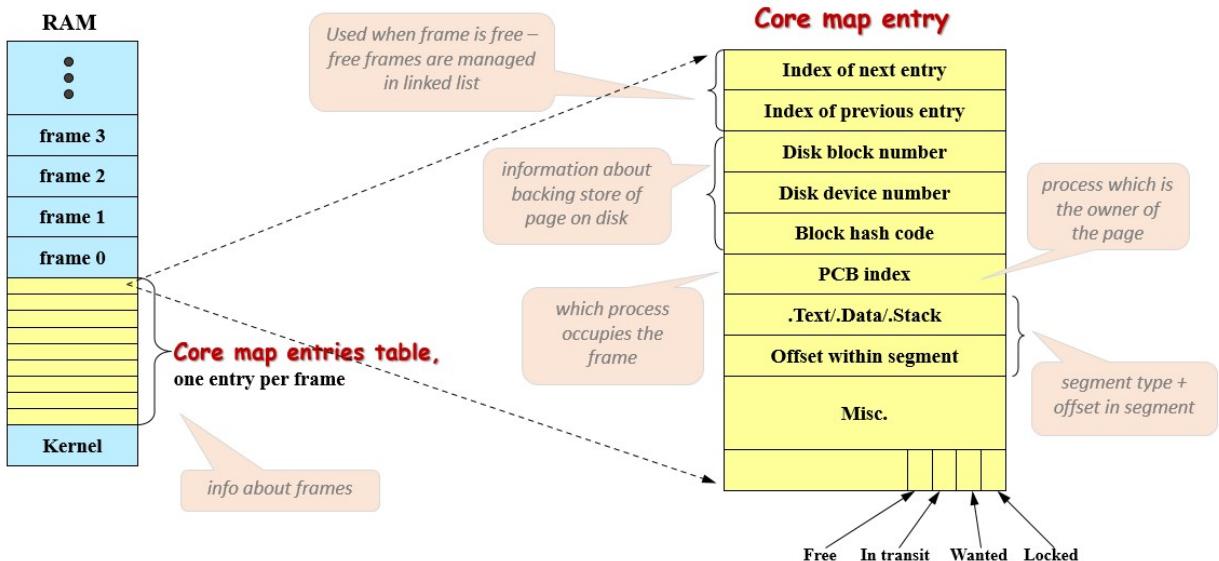


Unix system calls

- o `s=brk(addr)` – change data segment size
- o `a=mmap(addr,len,prot,flags,fd,offset)` – map (open) file *fd*, starting from *offset*, of length *len*, to virtual address *addr*
- o `s=unmap(addr,len)` – unmap a file (or a portion of it)

Unix 4BSD memory organization

Berkeley Software Distribution version 4



Unix Page Daemon Thread

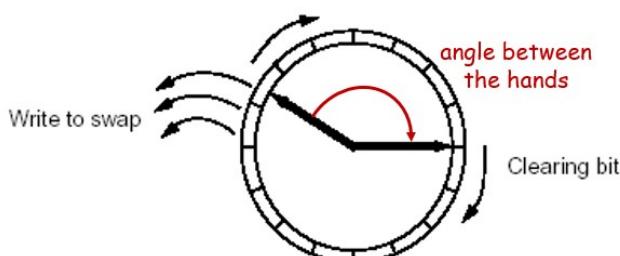
- freeing of page frames is done by a **page daemon** thread
- periodically inspect the state of memory - **if less than ¼ of page frames are free**, then it *frees page frames*

Unix page replacement algorithm

Two-Handed Clock algorithm

- first hand is used to clear R-bit
- second hand is used to free pages with R-bit=0
- “angle” parameter between the hands
 - small angle leaves in RAM only heavily used pages
 - i.e., if page is referenced before 2'nd hand comes, it will not be freed

OS Page daemon thread periodically inspect the state of memory. If less than ¼ of page frames are free, then it promotes two clock hands (first hand, and then second hand) to free some pages.



If two hands of the clock are close, then pages with R-bit 0 have only short time to survive in RAM – only if this pages are accessed in a short time.

Unix thrashing policy

- **On thrashing:**

- Remove processes idle for 20 sec or more
- If none – swap out the oldest process out of the 4 largest

- **Who get swapped back to RAM:**

- oldest swapped out processes first
- small memory-size processes first

ניהול זיכרון ב-XINU

מרחב הכתובות של תהליך ביוניקס

ביווניקס, לכל תהליך יש מרחב כתובות ייחודי, אשר כולל את הסטאק, ה-bss, ה-data וה-text. המצביע בראש המחסנית במרחב הכתובות הפיזיות שלו.

Memory-Mapped Files

הקבצים הממוספים בזיכרון (Memory-Mapped Files) הם אחד מהיבטים הייחודיים של ניהול הזיכרון ב-XINU. באמצעות שימוש בקריאות מערכת, ניתן ליצור התאמה בין מרחב הכתובות של התהליכים השונים ובין הקבצים בזיכרון. פעולות קראיה וכתיבה בקובץ בראם יכולות לשמש כאמצעי תקשורת בין תהליכים.

קריאות מערכת:

- א. brk - משנה את הגודל של סגמנט ה-data.
- ב. mmap - מפה את הקובץ fd לכתובת הירטואלית addr. המיפוי מתחילה מה-offset ויש לו אורך cha.
- ג. munmap - מפסיק את המיפוי של קובץ או חלק ממנו.

Unix 4BSD Memory Organization

בסביבת ה-XINU קיימת ארגון מיוחד לניהול הזיכרון, בו כל מסגרת זיכרון מנוהלת על ידי רשומה "Core Map". הרשימה מכילה מידע על הפירים, כולל סטטוס, מקום בדיסק, בעליהם של הפירים, סוג הסגמנט והיחס.

בנוסף, ברשומה PCB ניתן למצוא את האינדקס של ה-Control Block של התהליך. המבנה זהה דומה לו – previous entry-i next entry-Inverted Page Table.

Unix Page Daemon Thread

ת'רד ה-Paging Daemon ב-XINU אחראי לשחרור פיריים באופן תדיר. כאשר פחות מרביע מפיריים פנוים, התהליך הזה משחרר פיריים מהזיכרון. המטרה היא למנוע מצב של Thrashing. חשוב לציין כי הת'רד

Unix Page Replacement Algorithm - Two-Handed Clock Algorithm

אלגוריתם הוחלפה שב-UNIX נקרא "אלגוריתם השעון בעל 2 מנגינים". באמצעות שני מנגינים, כאשר הראשון מאפשר כיבוי של בית ה-R, והשני משחרר דפים שבית ה-R שלהם מכובה. כמו כן יש פרמטר נוסף - היזווית בין השניים, שבעזרתה ניתן להגדיר אילו דפים ישארו בזיכרון.

האלגוריתם עובד כך שהמנגין הראשון בעצם בוחר מועמדים להדחה מה-set והמנגין השני הלה למשחרר אותם. היזווית בעצם קובעת את אודל ה-set. ככל שהיזווית בין המנגינים קטנה יותר, קבוצת ה-set המשחררת אוטם. היזווית תהיה מרכיבת מדפים שהיו יותר בשימוש. הדבר המוביל במניעת Thrashing, אולם יש להיזהר מהזיהוי הלקי של פרופיל השימוש בזיכרון.

Unix Thrashing Policy

כאשר המערכת נמצאת במצב של Thrashing, היא מבצעת מדיניות כדי למנוע חריגה זו. היא תחילת מחפשת תהליכיים שהיו במצב המתנה לפחות 20 שניות. אם יש כאלה, היא מסירה אותם. אחרת, היא משחררת את התהליך הישן ביותר מtower ארבעת התהליכים הגדולים ביותר.

בעת החזרת תהליך לראם, יש עדיפות לתהליכיים שהיו היכי הרבה זמן באזור ההחלפה ועדיפות לתהליכיים בעלי הזיכרון הקטן ביותר.

Memory management, part 3: outline

❑ Segmentation

❑ Case studies

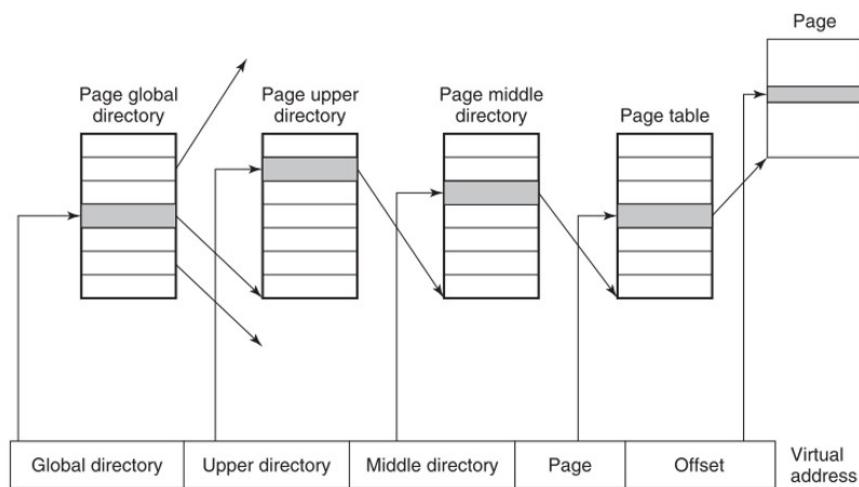
- MULTICS
- x86 (Pentium)
- Unix
- Linux
- Windows

Linux

- Each process gets **3GB** (of 4GB) **private virtual address space**
- Remaining **1GB** is occupied by **kernel** and **process page tables**

Linux page tables organization

- Expanded to **4-level indirect paging** in Linux 2.6.10.
- In Pentium architecture, the two middle levels are degenerated



Linux

עקרונות בסיסיים

הזיכרון הראשי בلينוקס מורכב מ-3 חלקים. החלק הראשון הוא הkernel, החלק השני הוא מיפוי הזיכרון. החלק השלישי הוא הזיכרון הפנוי לתוכניות, והוא מחולק למסגרות.

כפי שלמדנו, כל תהליך רץ במרחב כתובות ייחודי המחולק ל-3 סגמנטים: text, data, and stack. סגןמנט הטקסט הוא למשזה הזיכרון בו מאוחסן הוראות המכונה של התוכנית. סגןמנט זה נוצר על ידי תהליך הקומpileציה והאסmbלי והוא לקריאה בלבד. סגןמנט זה הוא סטטי וגודלו קבוע.

סמנט הנתונים, הוא הזכור בוא מאחסנים המשתנים של התוכנית, כולל מחרוזות, מערכים ומידע נוספת. הסמנט מחולק ל-2 חלקים: נתונים מאוחלים נתונים לא מאוחלים, כאשר מנסיבות היסטוריות, האחרון נקרא גם SSS.

סמנט המחסנת מתחילה בראש כתובות הירטואליות של התחליר וגדל מלמעלה למטה. ארכיטקטורה 32bit x86, סמנט זה מוגבל ל-3GB ומגבלה זה אינה מוגבלת על ידי תהליכי. למעשה מ透 4GB שמקצים לתחליר, התחליר מקבל 3GB ו-1GB הם לטובת הernal וטבלאות דפים.

ארגון טבלאות הדפים בלינוקס

ליבת הלינוקס מתאימה את מודל הזיכרון לארכיטקטורה של המחשב עצמו, אך באופן כללי לצורך ניהול זיכרון ייעיל גם ב-32Bit וגם ב-64Bit, לינוקס משתמש בטבלת דפים ב-4 רמות (בגרסה 2.6.11, ראה הערת העור), ככלمر כל כתובת מחולקת ל-5 שדות. כאשר ה-4 הראשונות הן לטובת איתור המיקום בטבלת הדפים ברמה الأخيرة והאחרונה בשבייל מציאת הכתובת הפיזית המבוקשת.

עם זאת, ארכיטקטורות כמו פנטיום, אשר תומכות רק ב-2 רמות, הרמה העליונה והאמצעית מכילות רק ערך אחד. באופן דומה, ניתן להפעיל מערכת ניהול דפים ב-3 רמות.

הערה:

בגרסה 4.14, יחד עם השכלולים שדור ה-skylake של אינטל הציג, לינוקס החלה לתמוך במערכת ניהול דפים ב-5 רמות.

מכניקות ניהול זיכרון

בלינוקס יש מספר כלים לניהול הזיכרון הראשי, אשר מנוהל לפי מדיניות ה-paging demand. בסיס נמצאת **מקרה הדפים**, אשר עושה שימוש באלגוריתם ה-buddy. עוד שימוש באלגוריתם זה הוא גם עבור הקצאות זיכרון עבור הקצאות זיכרון של דרייברים.

כפי שכבר למדנו, יש באלגוריתם פרוגמנטייה פנימית וכן לינוקס הציגה מכנים משלים: **SLAB allocator**. בעזרתו מכנים זה, לינוקס מנהלת את תא זיכרון הפנימי שנעשה בהם שימוש באלגוריתם ה-buddy, כאשר הוא מחלק אותם ליחידות קטנות יותר - דבר המקטין את הפרוגמנטייה הפנימית.

בנוסף, קיים מנגנון נוסף, הנקרא `vmalloc`, אשר נועד להקצאות זיכרון רציפות, במיוחד עבור מרחב כתובות וירטואליים גדולים.

אלגוריתם החלפת הדפים בלינוקס הוא וריאציה של אלגוריתם השעון ונקרא: **The Page Frame Reclaiming Algorithm**. האלגוריתם מבחין בין סוגי שונים של דפים וمبין כל הדפים המועמדים לכטיבה לאחסון, האלגוריתם מתעדף דפים אשר יהיה קל יותר להחזיר אותם זיכרון.

הערה: 2 החלקים הראשונים, מיפוי הזיכרון, והזיכרון של הkernel, אינם ניתנים להחלפה.

Memory management, part 3: outline

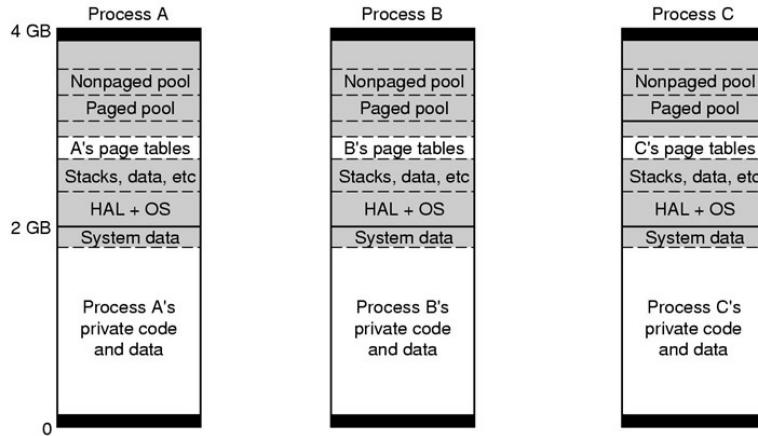
Segmentation

Case studies

- MULTICS
- x86 (Pentium)
- Unix
- Linux
- Windows

Win 8: virtual address space

- **2 GB for processes** – private for each process
- **2 GB for Kernel** - shared among all processes



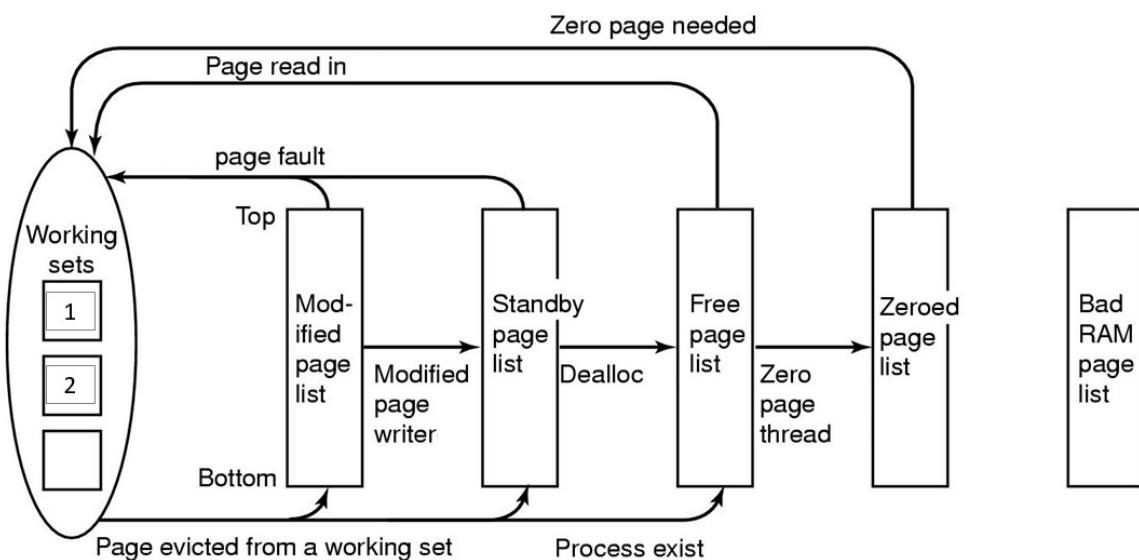
Win 8: memory management concepts

- Each **virtual page** can be in one of following **states**:
 - *Free/invalid* – currently not in use, a reference causes access violation
 - *Committed* – code/data was mapped to virtual page
 - *Reserved* – allocated to thread, not mapped yet.
For example, when a new thread starts, 1MB of process space is reserved to its stack.
 - *Readable/writable/executable*
- **Dynamic backing store management**
- **Supports memory-mapped files**

Win 8: page replacement algorithm

- Pages 'age-counters' are maintained
- Process has **working set** defined by two parameters – the minimal and maximal # of pages
- WS is **updated on each page fault** (*i.e., the data structure WS*) -
 - if WS < Max, then *add* page to WS
 - if WS \geq Max, then *evict some* page in WS
- **If a process thrashes, its working set size is increased**

Various page lists and transitions between them



ניהול זיכרון ב-(32 bit) Windows 8

הקצתת זיכרון

במערכת הפעלה Windows 8, הזיכרון הראשי מחלק לשניים: 2GB לתוכים ו2GB ל kernell. זיכרון זה מנוהל בצורה מדוקית ונקודתית על ידי המערכת.

מבנה הדף הווירטואלי

בכל תחילה במערכת קיימת הקצתה של זיכרון וירטואלי. הדף הווירטואלי מאופיין במספר מצבים שונים: אם הדף פנוי, האם קיים מיפוי קוד או נתונים לדף זה, או האם הדף הוקצה לתדרד שנוצר עכשו. כל דף זיכרון גם מוגדר עם הרשאות מיוחדות.

המערכת מתחזקת לכל דף ערך איל ולכל תהליך היא מתחזקת סט עבודה המוגדר על ידי שני פרמטרים: מספר הדפים המקסימלי והמיןימלי. הסט מתעדכן בכל page fault, וכאשר סט העבודה חורג את מספר הדפים המקסימלי, המערכת זורקת דף כלשהו. במקרה של Thrashing, הגודל של סט העבודהadel.

ניהול מקום פנוי

במקרה בו המערכת תשחרר זיכרון כאשר המקום הפנוי נגמר, היא תכתוב גם את הדפים "מלוכלים" אל הדיסק. ניהול מתבצע על פי מבני המידע שמאחסנים את פרטי הדפים בתהליכי המתוואר בתרשימים לעיל.

עזרה: במהלך ההרצאה עלתה סוגיה קלה. יש הבדל דק בין להbias table בשבייל הסטוק לבין ייצור כזה בשבייל תהליכי. הבאת טבלת דפים כרוכה בפגיעה באבטחה של מערכת הפעלה, שכן היא מאפשרת לתהליכי משתמש לקרוא מידע השמור ל kernell.



מערכת הפעלה 2022 ©

Operating Systems

Lecture 9 – Memory Management – Segments

Dr. Marina Kogan-Sadetsky

העשרה:

הש侃פים הנוגעים בניהול הזיכרון בוינדוס ככל הנראה מתייחסות לארסה יחסית ישנה. החל מוינדוס 11, ווינדוס כבר לא תומכת ב32 ביט. זאת בנוסף להבדלים נוספים במחודורה 5 של הספר של טננបאום. השינויים בגודל הם: יש 256 TB בסך הכל, כאשר חצי הולך למרחב המשתמש וחצי הולך לkernel. בעוד שזה נראה גדול מדי, ווינדוס שומרת נתק לא קטן מהחצى לkernel לאופטימיזציות ביצועים, מידע מסוים וכו'.

ב. היא תומכת בדפים בגודל 4KB וכן בדפים גדולים (בגודל 2MB) ובדפים ענקיים (בגודל 1GB), במידה והחווארת תומכת בזאת.

ג. מבנה תמונה התהליכי שונה וcutout נראה כך:

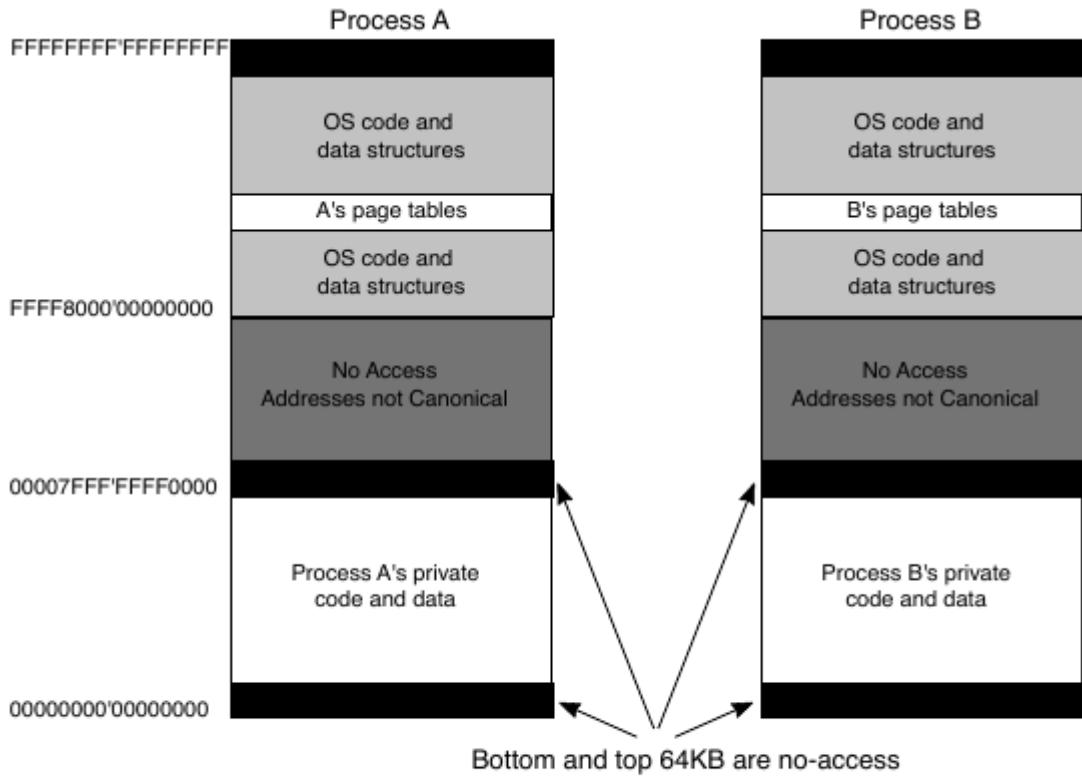


Figure 11-31. Virtual address space layout for three 64-bit user processes. The white areas are private per process. The shaded areas are shared among all processes.