

הרצאה 1 – מבוא

מטרת הקורס: מה היא מערכת הפעלה? נverb על רכיבים שונים במערכות הפעלה, גם מערכות הפעלה מסורתיות ואייר הן משפיעות על מערכות הפעלה מודרניות. מערכת מחשב – איך צריך לבנות מערכת מחשב, איך לארגן אותה. עקרונות על פיהם נבנו מערכות הפעלה מודרניות ופחות מודרניות.

☰ מנהלות:

- 65% בחינה סופית, 35% משיעורי הבית (עד חמישה תרגילים, ארבעה חובה ואחד רשות).
- התרגילים יכתבו ב-C, C++.

מה היא מערכת הפעלה?

users
applications
operating system
Hardware / storage

מה היא מערכת מחשב? מה הרכיבים המרכזיים של מערכת מחשב?

מקובל להסתכל על מערכת מחשב במבנה של שלושה חלקים לוגיים:

1. **Users** – המשתמשים במערכת המחשב.
2. **אפליקציות** – כל תוכנית שרצה על חומרה, מורצת ע"י היוזרים.
3. **חומרה** – כוללת את המעבד – CPU, גישה לאחסון – Storage, זיכרון. עליה רצות האפליקציות.

מערכת הפעלה – המערכת דרכה אפליקציות יכולות להשתמש ולתקשר בצורה יעילה עם חומרה ועם שאר התקנים במחשב. נונטן לנו את התשתית לדבר ישירות עם ה-CPU.

- חוצצת בין היוזרים והאפליקציהקציות לחומרה והאחסון.
- מגיבה להוראות מהאפליקציהקציות ומעבירה לחומרה, או מעבירה אירוע מהחומרה אל האפליקציה (לדוג' סיום הצגת האות על המסך, סיום הדפסת הדף).
- לא נועדה לבצע פעולות مثل עצמה, מאפשרת ביצוע פעולות של האפליקציות.
- מערכת ההפעלה תמיד רצה, תמיד ברקע. אי אפשר להריץ אפליקציות ללא שימוש במערכת הפעלה.
- מנהלת את המקובלויות של התוכנית – כמה יוזרים יכולים להשתמש וכמה אפליקציות יכולות לרוץ ללא התנגשויות. מבודדת בצורה בטוחה ונוחה.

אם היינו רוצים לבנות תוכנת מחשב ללא מערכת הפעלה:

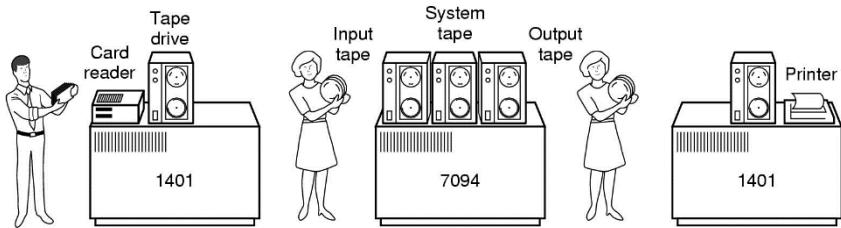
- אם היינו מאחסנים את התוכנית במקום ספציפי בזכרון כי ה-CPU יתחיל לחפש ב-0. בעצם היום מערכת הפעלה מורה ל-CPUs באיזו כתובת התוכנה מתחילה.
- אם היינו רוצים להשתמש בעבר/מקלחת וכו' היינו צריכים לתקשר ישירות עם החומרה. אם היינו מחליפים מקלחת/עכבר היינו צריכים לבנות את התוכנית מחדש.
- אם היינו רוצים לגשት מקום מסוים בזכרון היינו צריכים להגיד לחומרה לבדוק מה לעשות כדי הגיען ל זיכרון.
- אין בדיקה שמה שעשינו בסדר וניתן כבה להרים את החומרה.
- לא ניתן לקיים מקובלויות כי אין מה שינהן זאת.

מערכות הפעלה **אפשרות אבסטרקציה**, לדוגמה קובץ – אבסטרקציה לאחסון מידע בדיסק.

ההיסטוריה של מערכות הפעלה

הדור הראשון 1945-1955 – המחשבים הראשונים, ללא מערכות הפעלה בכלל. מערכות הפעלה הראשונות הופיעו בשנות החמישים.

1955-1965



בצד אחד יש קלט שמיشهו מכנים ובסופו יש את הפלט – printer או ברטיס ניקוב. בעצם תכניות היו נכתבות ע"י ניקוב ברטיס. את הברטיס היו מכניםים לקרוא ברטיסים (1401), הוא היה כותב את המידע על סליל מגנטית ומישהו היה מעביר את הסליל לתוך המחשב (7094) שהיה עושה את החישובים. היו מעט מחשבים כאלה והם על הרבה.

אופרטור – כדי ליעל את התהליכים היה מפעיל/מפעילת מחשב. היוזר והאופרטור לא היו אותו אחד וזה אפשר התמקצעות.

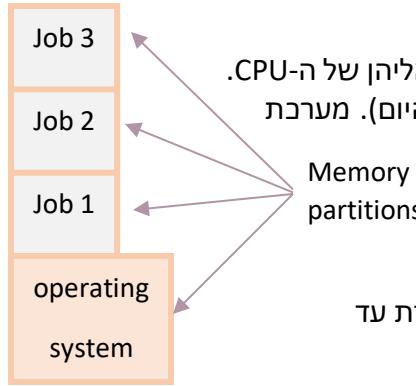
שמו בינה ברטיסים/תכניות אחד אחרי השני שבו נכתבים על טיפ אחיד והמחשב היה קורא אותם אחד אחד ומוציא טיפ אחד של פלט שמוביל את בולם, חסר זמן. אחד העקרונות הבסיסיים של מערכות הפעלה בימים.

כדי לבצע batching המחשב היה צריך להשתפר – לעבור אוטומטיות מג'וב לג'וב (מתוכנית לתוכנית), ולשם כך פותחה ה-**resident monitor** (7094) במקביל, להדפיס על חברה שהסתירה מהזמן שהמחשב עבד על עבודות חדשות למשל.

הבסיס לכך נקרא **Simultaneous Peripheral Operation On-Line – Spooling**. פייפליין שמאפשר הדפסה במקביל לחישוב, חישוב במקביל לקריאיה. יש sook job זהה מבנה נתונים שմוביל את כל מה שהסתירות, ומאפשר לרכיב הבא בפייפליין לבחור איזה ג'וב לבצע כדי להגברת היעילות של המעבד. אם זה לא היה, היה צורך לחכות עד שעבודה מסיימת תיגמר. הקונספט עדין משמש היום בהדפסות.

מונחים למדידת ביצועים של מערכת הפעלה:

1. **Latency (זמן שהייה)** – הזמן שלוקח לבצע מטלחה/ג'וב.
יחידת מדידה: שנייה ונדירותיה.
2. **Throughput (תפוקה)** – הקצב שבו מצלחים לעשות עבודה (למשל במקרים של מידע שמלוחים לעבד ביחידת זמן). יחידת מדידה: של byte/second.
3. **Utilization (ניצול)** – שבר. ביחס הזמן בה ה-CPU מבצע מטלות אמיתיות.
4. **CPU usage** – הזמן הכלול בו ה-CPU מבצע הוראות (גם משימות ייעילות וגם לא). תמיד יהיה גדול יותר מהניצול. לעיתים לא ניתן למדוד את הניצול ולבן נמדד את ה-usage.
5. **Overhead (תקורה)** – במקרים הפעולות העיקריות שהמערכת מבצעת כדי לבצע את המטלות. נמדדת להימدد בזמן/זיכרון/רוחב פס של תקשורת וכו'.

הדור השלישי 1965-1980

המחשב מטפל במספר תוכניות במקביל. התכניות נכתבו בזיכרון על מנת לאפשר גישה אליהן של ה-CPU. הזיכרון היה מחולק ובחלק אחד שלו בתובה התוכנית של מערכת הפעלה (בכונן עד היום). מערכת הפעלה הייתה שולחת ל-CPU פקודה לבצע אותה ג'וב.

Multi-programming – כל הג'ובים נמצאים בזיכרון ומבצעים אחד אחריו השני, כאשר עוברים אחד לשני כאשר תוכנית מסוימת אוזן כאשר היא מוגדרת על מערכת הפעלה. יותר על מערכת הפעלה יכולה לkerot באשר תוכנית ניגשת להתקן חיצוני שלוקח זמן (מקלחת, מסך, מדפסת). המערכת עוברת לטפל בתוכנית אחרת עד שהתקן החיצוני יסימן את הפעלה שלו כי בזמן זהה ניתן לעשות חישובים אחרים.

Time sharing – תוכנה יכולה לוטר על מערכת הפעלה או אם היא לוקחת הרבה זמן, מערכת הפעלה להוציא אותה, להכניס תוכנה אחרת במקומה ולהחזיר אותה אח"כ.

מעבד מריצ' תוכנה אחת, אין כמה תוכניות במקביל. מערכת הפעלה היא גם חלק מהתוכנות שרצות וכשהיא מבצעת פעולות מסוימת היא רצחה ושאר התוכניות לא. ה-useful tasks של מערכת הפעלה היא כל התוכניות באשר מערכות הפעלה עצמה היא התקורה, כל הפעולות שהיא מבצעת כדי לאפשר את הריצת התוכניות הן בתקורה.

- יש התקורה יותר גדולה ב-time sharing כי כל החלפה של ג'וב אחד בשני היא התקורה. ב-time sharing החלפות יזומות, גם בשלא מוגדרים, لكن יש יותר החלפות. למרות זאת משתמשים ב-time sharing, **response time**, יודעים שתוך א זמן מסוים יסתכל על התוכנית.
- מבחןת זמן השהייה, תלוי בתוכנית עצמה.
- הריצה של מערכת הפעלה היא בעצם שני ורב – המעבד יכול לבצע פקודה אחת בכל פעם, והוא מבצע או פקודה של מערכת הפעלה או של התוכנית אותה הוא מריצ'ה.
- איך תוכנית מועפת? פסיקת שעון – פעם בכמה זמן ממערכת הפעלה נכנסת לפעולה ומחליטה מי ימשיך אחרת.

תנאים (הנחות) המאפשרים את הריצות:

- מניחים שתמיד **יש CPU** ושהוא **תמידעובד**. אם לא משתמשים בו זה מבזבז זמן שועון. נרצה למקסם את ה-usage ואת ה-utilization.
- פעולות מבוצעות מהר יותר דרך ה-CPU** מאשר דרך התקן חיצוני. אם זה לא היה נכון לא הייתה סיבה לוותר על ה-CPU בשעורים להתקן חיצוני.
- דרישה שהזיכרון (RAM)** יהיה **מספק גדול** כדי לאפשר בתיבת של כל התוכניות בו.
- (DMA)** – **direct memory access** – ההתקנים החיצוניים יכולים להתנהל ישירות מול הזיכרון כדי לאפשר **multi-programming** ו-**time-sharing**. היום **יש DMA**.
- יש יותר ממשימה אחת** לבצע.

בדי לאפשר את ההנחות, יש צורך בפתרונות הבאים במערכת:

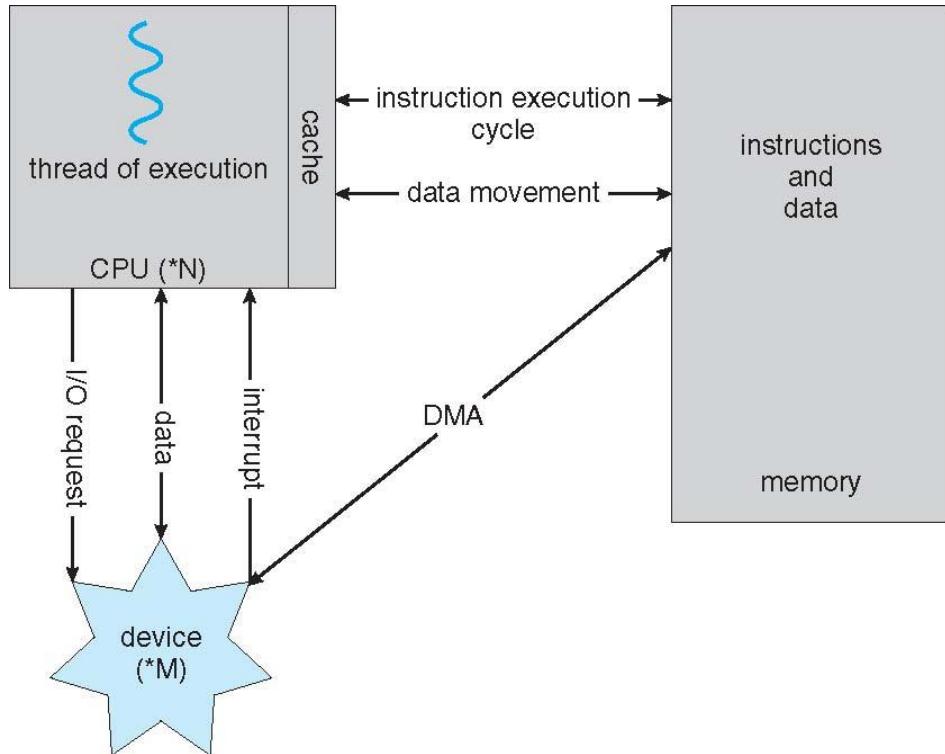
- תקשר עם התקני O/I** של מערכת הפעלה.
- ניהול זיכרון** בין הג'ובים שרצים ומ니עת דרישת תוכנית אחת לא תוכל לכתוב בזיכרון של אחרת.
- Scheduler**, מחליף מי הג'וב הבא שנכנס. ובנוסף, במקורה של time sharing, מתי להוציא ג'וב.
- Spooler**, מנהל את פעולות ה-O/I היותר איטיות (מה להדפיס קודם קודם למשל).

המחשה גרפית: שלושת הרכיבים המרכזיים בחומרה הם CPU (נדבר לעומק על המבנה מאוחר יותר), הזיכרון שמכיל את ההוראות ואת הדאטा.

ה-CPU קורא את ההוראות מתוך הזיכרון, מבצע חישובים וכותב אותם בזיכרון.

בנוסף יש התקנים devices – כל החומרה שהיא לא הזיכרון (מקלחת, עכבר וכו').

דרך ה-device drivers CPU כותב ההוראות ל-device. ברגע שה-device מבצע את הפעולה שלו הוא כותב את התוצאה לתוך הזיכרון דרך ה-DMA ומודיע ל-CPU שהוא מפסיק לבתוב interrupt (דבר על הכל בהמשך).



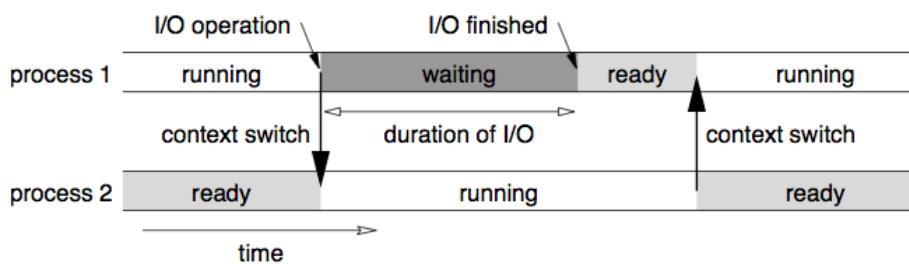
Processing תהליכי – אבסטרקציה של ג'ובים. לתחביבים יש שלושה מצבים עיקריים:

.1. Running - רצים על ה-CPU.

.2. Waiting - מחכים שימושו ב-IO/I/O קרלה.

.3. Ready - מוכנים, מחכים לתורם ב-CPU.

בדוגמה הבאה לביצוע, יש שני ג'ובים. בהתחלה 1. process במצב של ריצה, המעבד מבצע פקודות שלו, עד שmagieva פעולה של ה-IO/I. ברגע שהפעולהmagieva הוא עבר למצב של waiting, מהבה שפעולות ה-IO/I תיגמר, ומפנה את ה-CPU ל-2. process שיעבור למצב ריצה. בזמן זהה פעולה ה-IO/I הסתיימה ו-1. עבר למצב של ready. מתיישה 2 מסיים את הריצה, הופך ל-ready, ו-1 הופך ל-running. הריצה כזו היא time sharing, כי 2 לא וויתר בעצמו על הריצה, אלא מערכת ההפעלה החליטה על ה切换 בהרצה. הוא לא עבר מ-waiting ל-ready.



מערכות הפעלה היום (הדור החמישי)רכיבים עיקריים

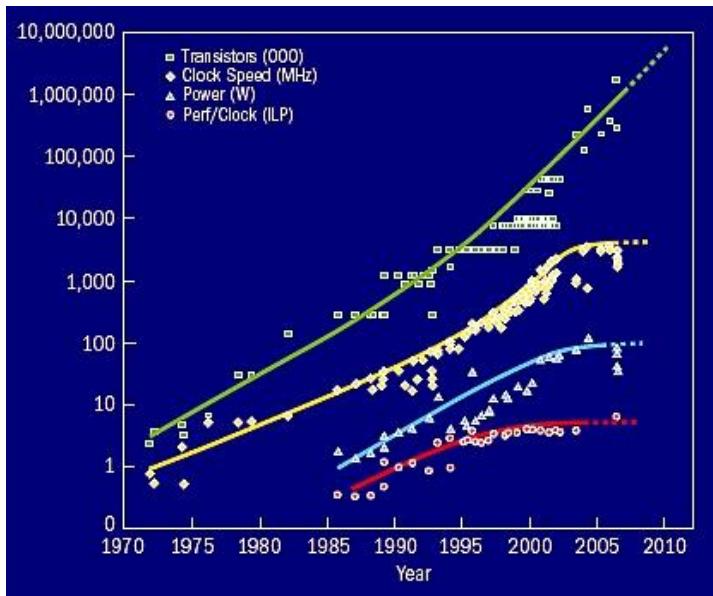
- ביצוע פעולות
- ניהול תהליכיים במקביל
- ניהול זיכרון
- תקשורת מול התקני I/O ודריברים
- מערכות קבצים
- שימוש ב-CLI/shell/GUI
- תקשורת בין מחשבים שונים
- דברים נוספים עליהם נדבר פחות: הגנה, cyber security, התמודדות עם טעויות ושגיאות, resource allocation וכו'.

בעצם מה שדיברנו עליו עד כה מיחס לדור השלישי ולדור הרביעי. בעצם, בדור החמישי, קיימים שני שינויים ארכיטקטוניים:

1. יש כמה מעבדים בלבד, כמה תוכניות רצות על כמה מעבדים. בקורס נתעסק בעיקר במעבד יחיד.
2. וירטואלייזציה – מכונות וירטואליות, VM, קונטינרדים.

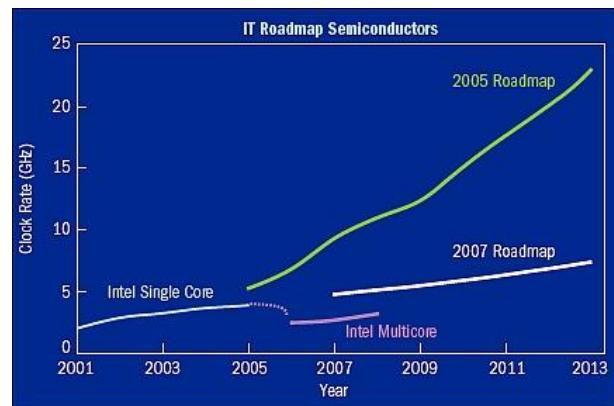
אפליקציות חדשות במערכות הפעלה:

1. הענן.
2. מערכות הפעלה ל-*mobile computing*.

Multi core

חוק מור – הומצא באינטל, היה נכון עד שנת 2005. לפני החוק כל מה הקשור למחשב (מהירות השעון, מהירות המעבד, כמות הטרנזיסטורים, כמות מידע וכו') עולה בצורה אקספוננציאלית, מכפילה את עצמה תוך שנה-שנתיים. זה אכן קרה עד 2005 שם הגיעו למחסום. מהירות השעון עולה אבל לא אקספוננציאלית.

עדין צריך לבצע חישובים מהירים, אז איך פיצוי על זה? במקום לשימוש מעבדים מהירים, שבו כמה מעבדים במקביל. מ-2005 החלו לפתח ארכיטקטורות של Multi-core וסדר העבודה עולה.



זה יוצר אתגרים מבחינתיים למערכת הפעלה:

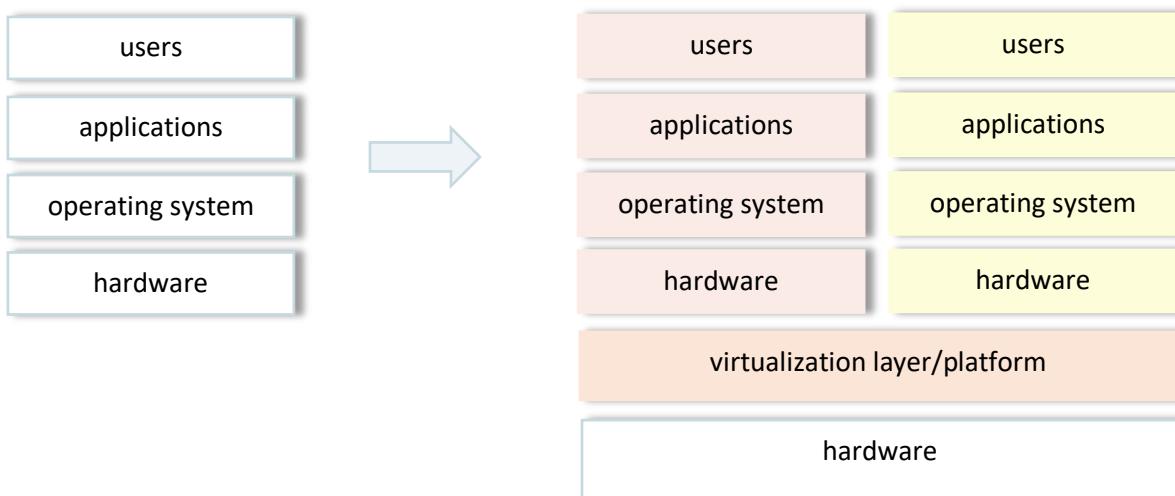
- חלק מה-resources משותפים בין המעבדים וחלק לא.
- ניהול זהה מורכב.
- ניסיונות שונים של איך מושגים את המקבילות: כמה מעבדים, כמה ליבות, וכו'.

חשיבות להציג שבסדיירנו על time sharing ועל multi-programming הראה במקביל היא לא אמת במקביל. ג'וב אחד רץ כל פעם, הרצות הן לסייעין.

וירטואלייזציה

לעומת השרטוט הקודם של היררכיות ברכיבי המחשב, יש מערכת הפעלה ווירטואלית שומריצה סוג של תוכנה שמדמה חומרה. מתחתיה יש **virtualization layer/platform** שמרתגם את החומרה הוירטואלית לפעולות שבוצעות על החומרה אמיתית.

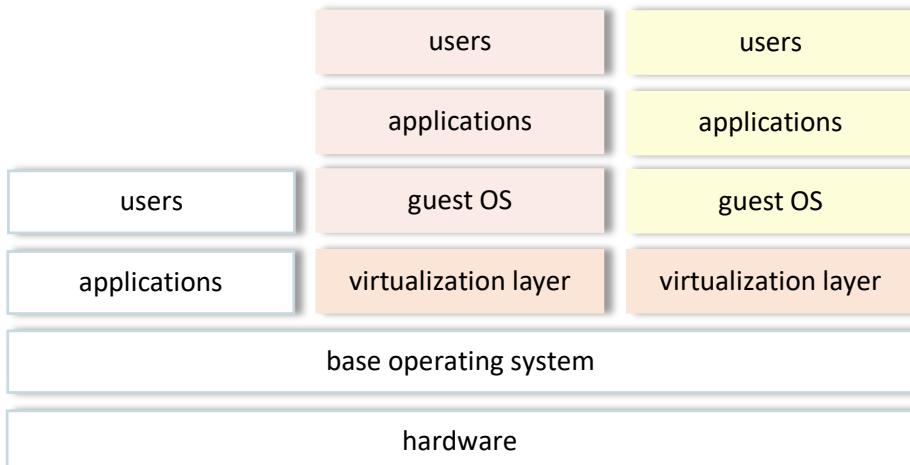
התкова של הוירטואלייזציה היא בתרגום של ה-layer לזכרון, זה שמבצעים את הפעולות בצורה עקיפה. היום ה-CPU תומך בזה והתרגם לעיטים מבוצע בחומרה.

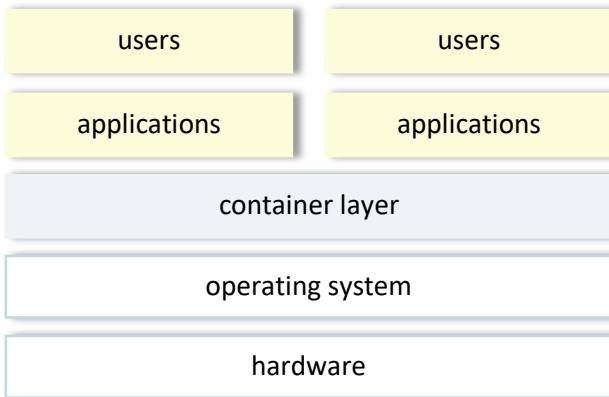


את התוכנה הוירטואלית ניתן להעתיק על מחשב אחר בלי להתקין ויש לזה יתרונות כמשמעותם בעיקר בתוכנות מעלה ענן.

הוירטואלייזציה מגיעה בכל מיני צורות:

- הרצה של מערכת הפעלה אחרת.





- קונטינרים – אין מערכת הפעלה בתוך מערכת הפעלה, אלא מערכת הפעלה מקבלת קונטינר שמכיל אפליקציות שרצות ביחד.

פתרונות של ווירטואלייזציה:

- .1 VM snapshot – להסכתה מה המצב של מערכת הפעלה.
 - .2 VM migration – העתקת המידע ממערכת הפעלה והרכתו במקום אחר.
 - .3 Scaling – אם מעוניינים ב-resources נוספים ניתן ליצור VM נוספים, שימושו להרצה בענן. אפשר גמישות.
 - .4 Isolation – הבדוק בין הג'ובים יותר טוב.
- יש תקווה בביצועים, עוד שכבה של מעקף.

הען – הרבה שירותים ומשאבי מחשב נדרדים דרך האינטרנט. אפשר לחסוך בחומרה במחשב האישי ולהשתמש בחומרה של מחשב מרוחק בצורה יعلا. כדי לבצע זאת משתמשים בוירטואלייזציה כדי לשתוף משאבי בין כמה יוזרים.

Mobile computing

	Desktop	Mobile Computing
Power	Electricity	Battery
Network	LAN	WiFi / Cellular
Main Memory	8-64GB RAM	3GB
Disk	TB HardDisk 512GB SSD	8/16/64 GB Flash Memory
CPU	Multiple Processors Cores	Slower and Smaller CPU & Cores
Input	Mouse / Keyboard	Touch Screen
Weight	No main concern	Concern

$$\text{Kilo} = 10^3, \text{ Mega} = 10^6$$

$$\text{Giga} = 10^9, \text{ Tera} = 10^{12}$$

- הפעלה ע"י חשמל במחשב לuemota בטרייה בטלפון משפיעה על ה-resources. האם ניתן להניח שה-CPU הוא בחינם? בموבайл, אם יש מעט בטרייה ניתן לשמר אותה לאחר בר. לא צריך להשתמש ב-UP-CPU במה שיותר.

אילו מההנחה על החומרה תקפות במקרה של mobile computing ?

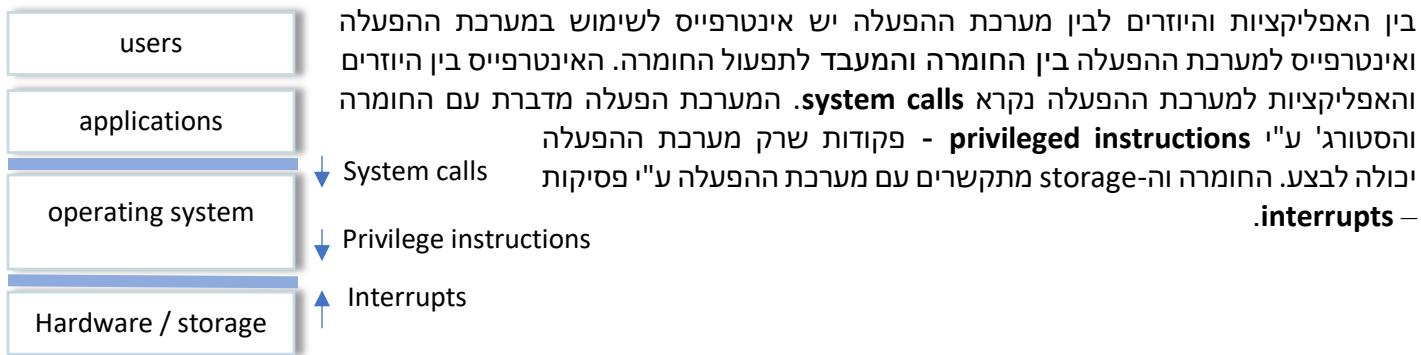
- | | |
|---|--|
| ✗ | תמיד יש CPU ושהוא תמידעובד – לא תמיד משתמשים בו. |
| ✓ | פעולות מבוצעות מהר יותר דרך ה-UP-CPU מאשר דרך התקן חיצוני. |
| ✗ | הזיכרון (RAM) יהיה מספיק גדול כדי לאפשר כתיבה של כל התוכניות בו – לא כזה גדול כמו במחשב. |
| ✓ | Direct memory access (DMA) |
| ✗ | יש יותר ממשימה אחת לביצוע – לרוב יש ממשימה אחת להרצה. |

קריטריונים של מערכות הפעלה של המובייל:

- קלים לתפעול למפתחי אפליקציות.
- זמן תגובה מהיר.
- בעליים מבחינה אנרגטית, צריכים לחשב על ניצול נכון של ה-UP-CPU.
- בטיחות חשובה מאוד.
- התחשבות בכמות זיכרון מצומצמת.

הרצאה 2

System calls



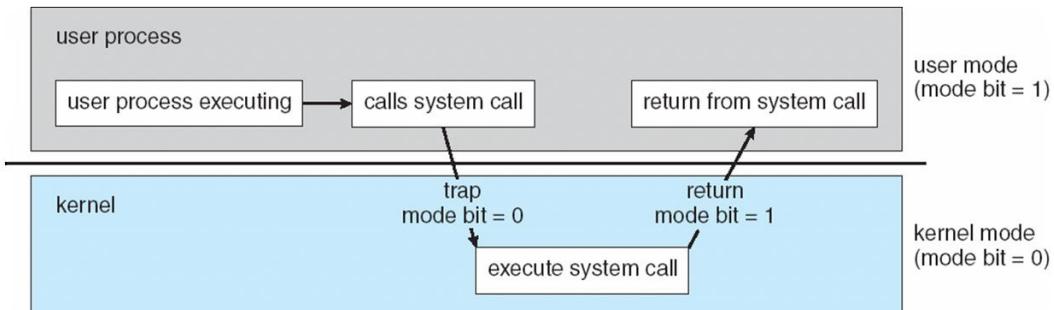
שני מצבים ל-CPU:

1. **User mode:** הפעלה מבצע קוד של אפליקציות ומשתמשים. מצב הריצה של המשתמש, לא יכול לגשת שירות לחומרה (אלא ע"י system call ומעבר ל-kernel).
2. **Kernel mode:** הפעלה מתקשרות עם מערכות הhardware (storage, memory, CPU). ביצוע פקודות של מערכות הhardware, מצב ריצה של מערכות הhardware.

- מעבירות מ-user ל-kernel, האינטראפיס בינם. התוכניות של היוזר מקבלות את השירותים של מערכת הhardware ע"י קרייה ל-system calls.
- דוגמאות לפונקציות, אבל שונות לחלוון מאחוריו הקלעים. כאשר קוראים ל-system calls נתונים למעבד שליטה על המערכת, מערכת הhardware מרים את הקוד שלא תור תעודף יוזרים אחרים ותקשר עם חומרה מסוימת.
 - ניתן לגשת לבני נתונים ולדברים הפנימיים של מערכת הhardware (מערכת קבצים, בני נתונים וכו') ע"י ה-system calls.
 - בודקות שהפרמטרים תקינים.

מעבר מ-kernel ל-user

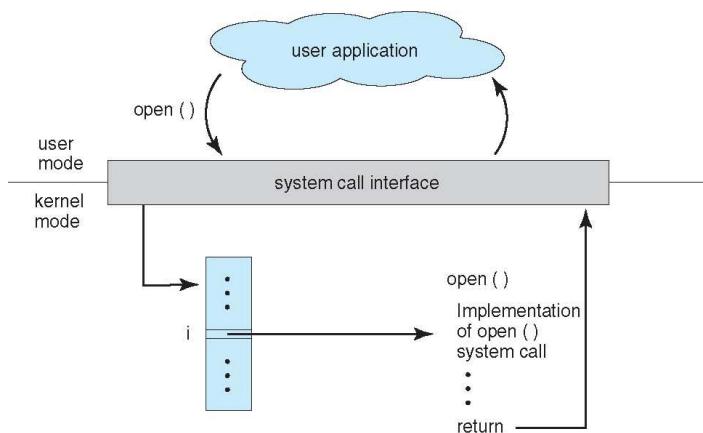
- התרשימים מתאר ריצה של מערכת הhardware. בהתחלה מריםם תהליכי של היוזר ובאשר קוראים ל-system call היוזר מועתר על המעבד – **trap** וועובר ל-kernel mode – בטרם שהיא מסיימת את מחזירה את השליטה לווזה מה שצורך, היא יכולה לבצע את הקוד ב-user mode, ברגע שהיא מסיימת את מחזירה את השליטה לווזה.
- בטור ה-CPU יש גרסה עם בית שמורה על איזה מצב עובדים ברגע. ה-UCPU בודק שהוא אכן ב-mode – בטרם שהוא שיבת למוד הנכון. אם לא המערכת זורקת exception.



לפעמים נעשות טעויות וקורה דבר שקורא ל-trap רק שהוא לא מתור הקוד. למשל חילוק ב-0 יעביר ל-mode exception שם יש התמודדות של מערכת הפעלה עם השגיאה (למשל זריקת exception).

סוגים של system calls

- **Process control** – יש הרבה ג'וביים שימושיים לסירוגין באותו מעבד כדי לבצע את הריצה. צריך לניהל את זה – ליצור תהליכים, לסיים תהליכים, לדעת איך לחזור לתהיליך הקודם כאשר עברנו ממנו לתהיליך אחר וכו'. זה עשויה ע"י קריאות של מערכת הפעלה.
- **File management** – ניהול קבצים, אבסטרקציה של מערכת הפעלה מספקת כדי לניהל מידע.
- **Device management** – ניהול התקנים חיצוניים (מקלדת, עכבר ועוד). קריאה מהם, כתיבה וכו'.
- **Information maintenance** – מידע שישיר לכל המערכת, למשל האם מצב כל-devices בסדר?
- **Communication** – תקשורת בין אפליקציה אחת לאפליקציה אחרת, בין אפליקציה אחת למחשב אחר.
- **Protection** – מי יכול להריץ אפליקציה מסוימת? מי יכול לקרוא מקובץ מסוים? למשל פקודות chmod.



Application programming Interface (API)

פקודה אחת שמבצעת הכנות לקלט ועיבוד לפلت במקום כמה פקודות אחרות.

למשל בכתיבה למסך ה-API הוא `printf`. הקריאה למערכת הפעלה עצמה היא ע"י `write`, שהקלט שלו הוא בתים. `printf` יותר עשירה ודרבה קודם עושיםprocessing לקלט ואז מבצעים את הפקודה `write`, ואח"כ ניתן לעשותprocessing בכיוון הפוך.

דוגמה נוספת (בشرطוט) היא הפקודה `fopen`. היא system call שבסופו של דבר מפעצת הפעלה ב-kernel mode תבצע את הפתיחה של הקובץ ולאחר מכן תחזיר לך את השליטה.

הרבה פעמים לא משתמשים ישירות ב-system calls אלא ב-API, למה?

- **אבסטרקציה** (יותר נוח להתנהל עם מחרוזות מאשר עם בתים)
- **בטיחות** (בדיקה שהקלט בסדר, ניתן להחזיר העורט לפני שעושים דברים חריגים)
- **ניידות** (ניתן לבצע בר-מעבר ממפעצת הפעלה אחת לאחרתבי-h-API לא משתנה, אבל ה-system calls ב-*c*)

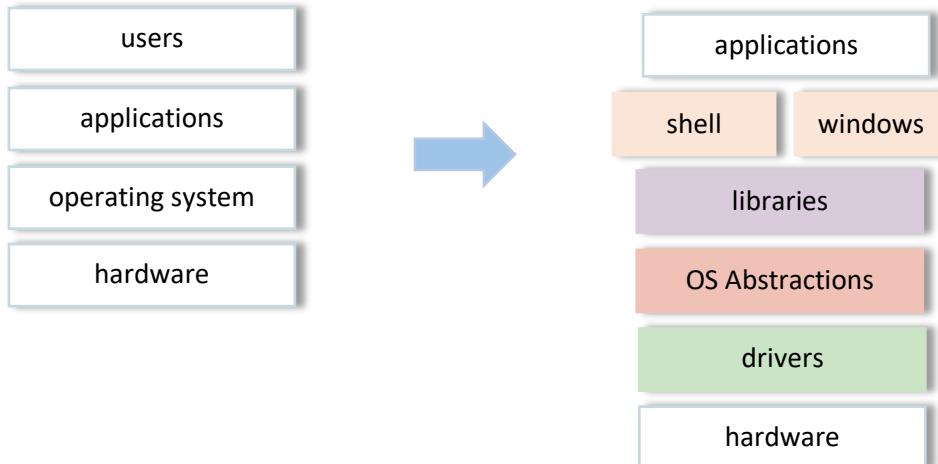
System programs

```
dhay — bash — 58x6
Last login: Sat Mar  5 13:49:01 on ttys000
Davids-MacBook-Air-2:~ dhay$ echo operating-systems
operating-systems
Davids-MacBook-Air-2:~ dhay$
```

מערכות הפעלה המודרניות מגיעות עם אפליקציות مثل עצמן שאפשר להריץ וככבה להשתמש בהן, ואלה אפליקציות הנונטנות שירות לאפליקציות אחרות. דוגמה: הפקודה echo כותבת ב-shell את מה שכתבנו אחרת. ב-*c* פקודה זאת לוקחת 274 שורות תוך שימוש בפונקציות ספרייה (`fprintf`, `fpwrite`) שעושות בעצם שימוש ב-system call, בר-מעבר של דבר יש שימוש ב-user mode echo וצ-ב-user mode echo ש-echo קורא ל-write הוא עשה `trap`, נתן למערכת הפעלה לבצע את write, ואז חזר. יש בפקודה הזאת כמה מעברים מ-user mode ל-kernel mode.

System programs זה חלק ממפעצת הפעלה אבל לא באופן מוחלט. הקווים שראינו בשרטוט מעלה שמנגנים את מערכת הפעלה הם לא מוחלטים.

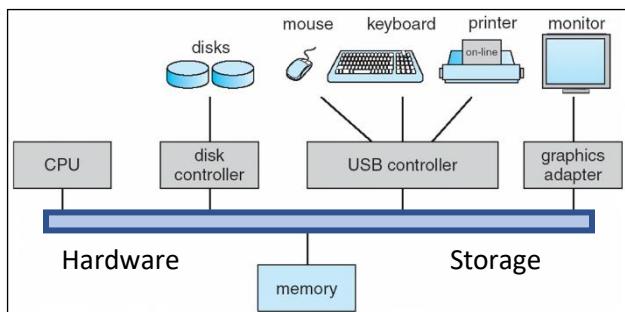
מתקבל להסתכל על מערכת הפעלה בבעל מספר שכבות:
השכבות התחתונות קרובות לחומרה והעליונות לאפליקציות (ה-system programs).



- צורך בתוכן system programs באופן נגיש למשתמש.
- האם system programs הן חלק מערכת הפעלה או חלק מהאפליקציות? גבול אפור, לא מוחלט.
- למשל web, browser, בום זה לא נחשב כחלק מערכת הפעלה.
- **הספריות** הן האינטראפיס שמערכת הפעלה נתנת כדי שייהי אפשר להשתמש בשירותים שלה ע"י האפליקציות אחרות (stdio ו-stdlib בס', java.net ו-Java.io בג'אווה).
- מתחת לספריות יש את **האבסטרקציה** שמערכת הפעלה נתנת: קבצים כאבסטרקציה לאחסון; חישוב ב-CPU שמתבצע ע"י תחilibים וזה אבסטרקציה לזה שכמה תחilibים רצים במקביל; תקשורת בין כמה מחשבים (sockets).
- **הדריברים** הכו קרובים לחומרה. הם בעצם תוכנה שנותנתת את האינטראפיס הספציפי בו ניתן לגשת לחומרה.

חיבור התקני O/I למערכת

Bus – קו תקשורת המחבר את התקנים למערכת. רץ עליו המידע וכל אחד מהתקני הקלט פלט יכולים להתחבר אליו וכן עבר מידע בין ה-device אל ה-CPU או אל ה-RAM (DMA).



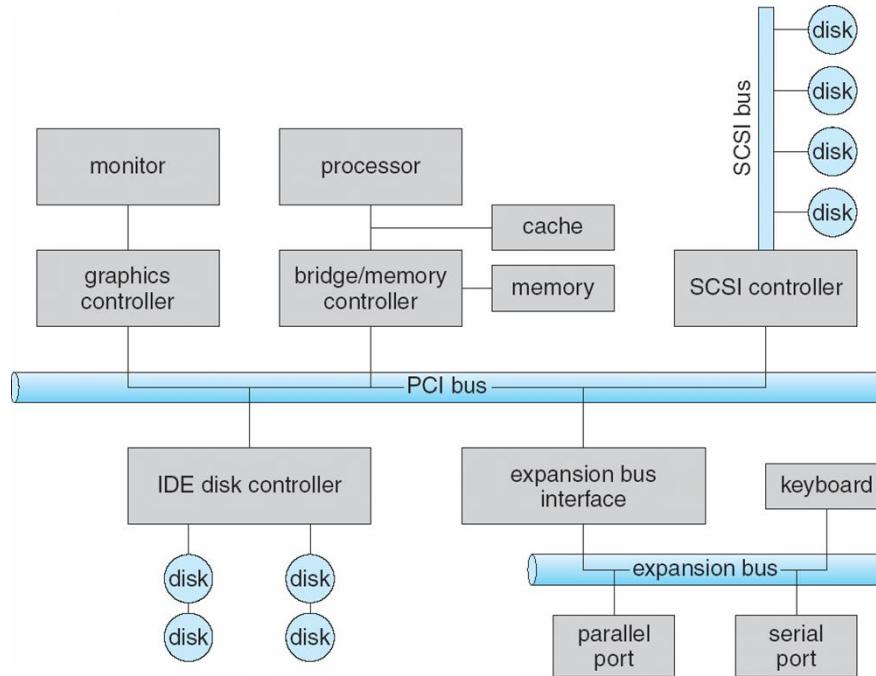
היתרונות:

- זול.
- ורstyלי, חיבור בינה devices ל-bus אחד.

החסרונות:

- צוואר הבקבוק של המערכת, מעביר מידע בצורה מוגבלת.
- הגבלה על כמות devices שאפשר לחבר.
- אמרור להתאים לכל הסוגים וזה מוגבל.

בפועל המצב מורכב יותר ויש כמה buses שכל אחד נועד לסוג ספציפי ונמנעות התנגשויות.



Interrupts

הדרך בה התקנים מתקשרים עם ה-CPU ודרךם עם מערכת הפעלה. התקנים רוצים לעדכן את מערכת הפעלה במקרים מסוימים, כמו שהם סיימו לבצע פעולה מסוימת או שקרה להם משהו.

המקרה בו הם יכולים לऋוץ בלי קשר לריצה (למשל לחיצה על כפתור).
דוגמאות ל-interrupts:

- **בקשות של החומרה**
- **התקני קלט פלט** – בלחיצה על כפתור במקלדת התוכן צריך להישמר בזיכרון וזאת ע"י DMA או ע"י שליחה ב-*bus*. לאחר מכן המקלדת מודיעה דרך קו תקשורת מיוחד מה – שמעביר סיגナル למערכת הפעלה שקרה משהו.
- **Page faults** (נלמד בהמשך)

יש חומרה מיוחדת במערכת שiodעת לטפל במקרים האלה ונראית *interrupt request line* שדרךה הם עוברים עד שהמעבד מקבל את הסיגנל.

בשא_INTERRUPT מתקבל בחומרה, ה-CPU יודע להפסיק לבצע את מה שביצע ולטפל ב-*interrupt*. לכל interrupt יש קוד, וה-CPU מבצע את רצף הפקודות הספרטיפיות שהן פקודות של מערכת הפעלה שמ恬בצעת ב-*kernel mode*. הקודים נמצאים ב-*Interrupt vector table (or interrupt description table)*.

משתנה בהתאם למה שכתוב בתוכנה ובזיכרון כתוב את הפקודות.

כשעוברים לביצוע של interrupt נדרש זכר לשוב לאן לחזור, שכאשר נסימן לטפל ב-*interrupt* ב-*kernel mode* נחזור לכתובת האחראונה אותה ביצענו. עושים disable לשאר ה-interrupts כדי שלא יהיה interrupt בתוך interrupt ורף קוד

של ה-CPU יכול לעשות את זה.

מה שגורם למערכת הפעלה לעשות פעולה הם בעצם ה-interrupt או ה-*system calls* (זהה גם סוג של interrupt).

hardware interrupts (התקני קלט פלט) או software interrupts (exceptions, system calls) יכול להתבצע ע"י interrupt (מערכת הפעלה).

לסיום, יש שלוש דרכים לתקשר עם מערכת הפעלה:

1. Hardware (external) interrupts
2. System calls (traps, software/internal interrupt)
3. exceptions

ובאופן כללי כלם נחקרים ל-interrupt, פסיקות, ומערכת הפעלה מונעת על ידם.

Sוגים של interrupts

1. תוכנה לעומת חומרה -

תוכנה: נגרים ע"י תוכנה, בתוצאה מביצוע קוד של יוצרים. מגיע מلمטה למערכת הפעלה. כולל exceptions system calls.

חומרה: מגיע מلمטה, מהתקני ה-I/O.

2. פנימי לעומת חיצוני -

פנימי: זהה לתוכנה. מתוך ה-CPU, כתוב ל-CPU מה לבצע או שהוא יודע בעצמו.

3. סינכרוניים לעומת אסינכרוניים -

סינכרוניים: יש שעון במערכת שמריץ את הפקודות וכל מחזור מתבצעת פקודה. interrupt סינכרוניים קורים יחד עם השעון.

אסינכרוניים: קורים במקום אחר עם שעון אחר.

4. – non-maskable
– maskable, ניתנים למיסוך לעומת disable
– Maskable – ניתן לעשות להם disable

Non-maskable – גם אם אנחנו באמצע ביצוע של interrupt אחר, נדרש עדין לטפל ב-interrupt
שאינם maskable.

5. **פרויודים** (מחזוריים) לעומת **אפריזדים** (קורים מתי שנרצה) – יש חומרה שקרובה ל-CPU שנקרה השעון, השעון עושה פעם בכמה זמן interrupt כדי להעביר את השילטה למערכת הפעלה, וזה interrupt פרויודי.

דוגמאות ל-interrupts

INT_NUM	Short Description PM [clarification needed]
0x00	Division by zero
0x01	Single-step interrupt (see trap flag)
0x02	NMI
0x03	Breakpoint (callable by the special 1-byte instruction 0xCC, used by debuggers)
0x04	Overflow
0x05	Bounds
0x06	Invalid Opcode
0x07	Coprocessor not available
0x08	Double fault
0x09	Coprocessor Segment Overrun (386 or earlier only)
0x0A	Invalid Task State Segment
0x0B	Segment not present

0x0C	Stack Fault
0x0D	General protection fault
0x0E	Page fault
0x0F	reserved
0x10	Math Fault
0x11	Alignment Check
0x12	Machine Check
0x13	SIMD Floating-Point Exception
0x14	Virtualization Exception
0x15	Control Protection Exception

לפעמים ה-user mode וה-kernel mode מתערבבים, כמו בדוגמה שראינו של הפקודה echo. כל מערכת הפעלה בוחרת איזה חלק ירוק באיזה mode.

בכל SHA-kernel mode קטן יותר:

- פחות traps ← פחות overhead ← תפקה יתר גדולה
- פחות שליטה על המערכת ופחות הגנה.
- Memory footprint קטן יותר.
- חלק גדול מהfonקציונליות של מערכת הפעלה נמצאת מחוץ ל-kernel ← אם הfonקציונליות משתנה, אין צורך בשינוי כל ה-kernel.

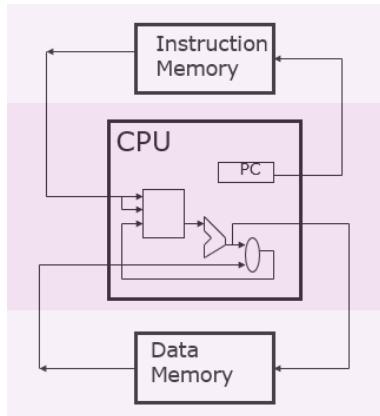
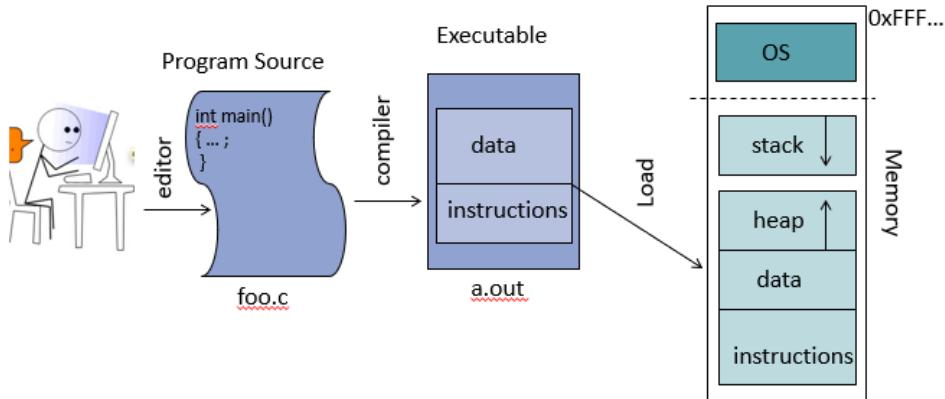
סוגים של kernels

1. **מונייטיים** – היה קיים בעבר, מערכת הפעלה אחת שכולה בזיכרון. אוטו.
2. **מודולרי-מונייטי** – יש את הfonקציונליות הבסיסית של מערכת הפעלה, ואם רצים להרחיב את הfonקציונליות ניתן להעלות מודולים לזכרון לצורך. במערכות הפעלה מודרניות.
3. **マイקו-קרנל** – יש כמה kernels שונים ומאוד קטנים שמשתפים פעולה.

הרצאה 3 – Process Management

איך מערכת הפעלה מרים תוכנות

מזכורת: לאחר בटיבת התוכנית ע"י היוזר, היוזר מكمפל את התוכנית והופך אותה ל-executable ← הקובץ עליה לזכור, ה-instructions והדאטה מועתקים לתוך הזיכרון מהקובץ ונפתחים מבני נתונים דינמיים של heap ו-stack ← התוכנית רצתה על ה-CPU.



המעבד מחלק לחלקים הבאים:

1. PC המצביע לתוכ זיכרון מיוחד בו בתיבות ה-instructions – פקודות אסמבלי.
 2. זיכרון שה-CPU ניגש אליו ישירות.
 3. גגיסטרים שונים.
 4. ALU המבצע חישוב אריתמטיים. מקבל קלט מתוך הרגיסטרים, פקודה לפועל, ומחזיר קלט שנכתב בזיכרון או באיזשהו גגיסטר.
- בשה-CPU מרים תוכנית עצם ה-PC מצביע לקוד של מערכת הפעלה או התוכנית, והערכים ברגיסטרים קשורים למערכת הפעלה או לתוכנית.
- ה-CPU מתקשר עם הזיכרון – ה-memory instructions memory ו-data memory.

Execution sequences

- שבירת הוראה שה-PC מצביע אליה ברגיסטר מיוחד בשם IR.
- ביצוע הפקודה – קריאה וכנתיבת לרגיסטרים/זיכרון.
- בתיבת התשובה.
- העברת הערך של ה-PC להוראה הבאה.

כל קורה בחומרה.

דוגמה: חיבור x בכתובות 100 עם y בכתובות 104

1. העלאת הערך בכתובות 100 בזיכרון לרגיסטר 1
2. העלאת הערך בכתובות 104 בזיכרון לרגיסטר 2
3. לחבר את רגיסטר 1 ו-2 ולכנתוב ברגיסטר 3
4. שבירת הערך שברגיסטר 3 בכתובות 100

```

lw r1, 100
lw r2, 104
add r3, r1, r2
sw r3,100

```

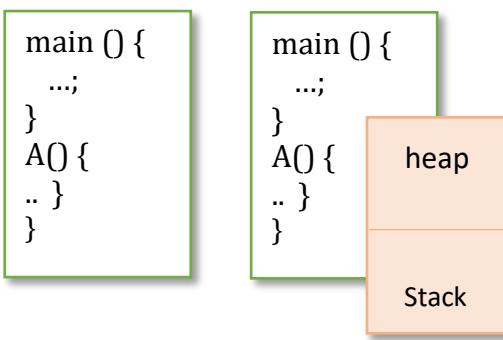
הקשר – Context (of execution)

הקשר מתיחס לערכי הרגיסטרים ו מצב הזיכרון בעת ביצוע התוכנית, מתאר את המצב בו ביצוע התוכנית נמצא.

- חלק מהמידע נמצא ב-CPU עצמו, ברגיסטרים השונים.
- מידע נוסף בקוד, הדאטה, ה-stack וה-heap נמצא בזכרון.
- דברים נוספים צריכים לנחל את זה ואלה שייכים למערכת הפעלה, גם הם כתובים איפשהו בזכרון.

תהליכים processes

תהליך process – אינסטנס דינامي המיצג ביצוע של תוכנית בהקשר מסוים.
מה שחשוב לגבי התהליך זה הקשר בו הוא נמצא. אם התהליך משתמש ב-CPU זה אומר שהערכיהם שלו כתובים ברגיסטרים וה-PC מצביע על ה-instructions שלו.
לפעמים באשר רוצים להפסיק ריצה של תהליך צריך לשמור את הקשר שלו כדי לחזור ולעבד עליו אחר כך.

תוכנית לעומת תהליך

תוכנית – רצף פקודות באסםלי, במצב סטטי.

תהליך – ביצוע של תוכנית. דינמי, עליה לזכרון, יש stack ו-heap.

- האם תהליך הוא יותר מתוכנית? אומנם תהליך הוא התוכנית עם הקשר, אבל לעיתים תוכנית אחת פותחת כמה תהליכים במקביל.

מרחב הכתובות של תהליך - address space of process

מרחב הכתובות – סט הכתובות בזיכרון אליו התהליך יכול לגשת.

בזמן הריצה לシリוגן של תוכניות נרצה למונע התנגשויות בכתובת בזיכרון, וכך בכל תהליך מוקצת מרחב בכתובות (אינטראול, רציף) מסויל שרק אליו ניתן לגשת.

מוגדר ע"י שני פרמטרים:

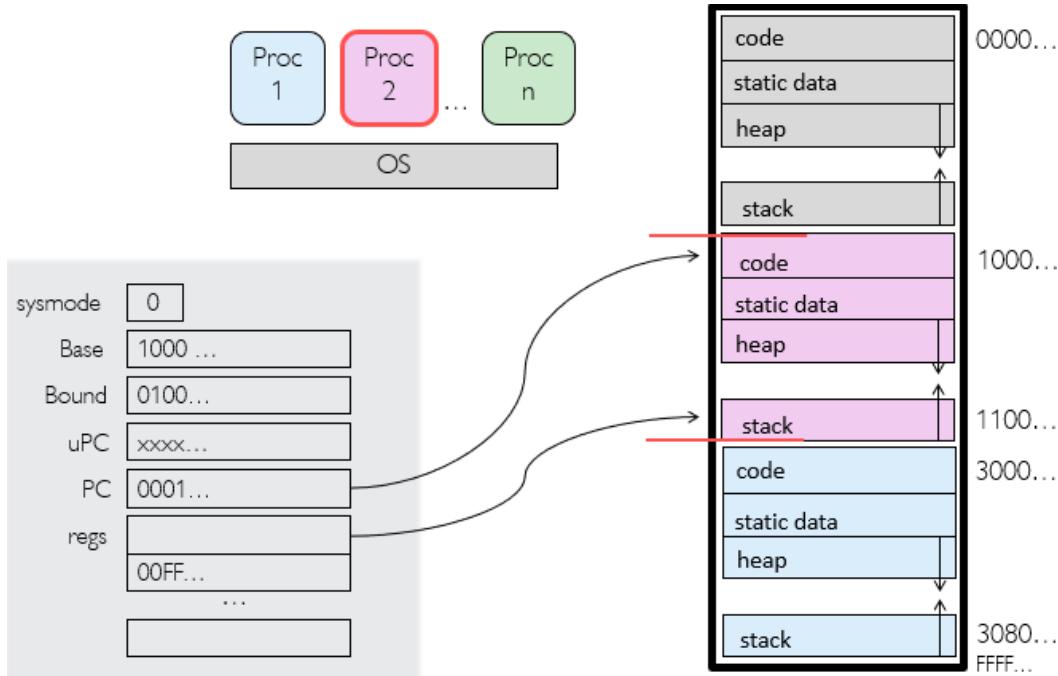
1. **Bound** - גבולות הזיכרון שמותר לתהליך לצרוך.

2. **Base** - כתובות התחלה ממנה מוקצת הזיכרון באופן רציף.

← לתהליך מותר לגשת לכתובות החל מ-**base** ועד **base + bound**. אם התהליך ינסה לגשת מעבר למרחב הכתובות, המעבד לא יוכל זאת ויזרק exception – exception תוכנה המורה על הפסקה ביצוע התוכנה. מאפשר הגנה ובודול של תהליכים.

← בשמתחילים להרץ process יש ריגיסטר מיוחד עם הערך של **base**, ה-CPU מתרגם את הפקודה שכוללת את **x = base + x** וכן לא ניתן לגשת לעולם לכתובת מתחת ל-**base**.

← כדי לוודא שאין חריגה מלמעלה קיים רגיסטר בו בתוכו ה-**bound**, וכל פעם שניגשים לכתובת **x** בודקים שהוא קטן מ-**bound**.



– התהיליך עליו ה-CPU עובד באותו רגע.

– הדיברונו מושתמשת מערכת הפעלה (באפור בצד ימין בשרטוט). כדי להריץ פקודות של מערכת הפעלה צריך שה-base וה-bound יצביעו לשם. יתכן מצב שבו הם יצביעו לחלק של מערכת הפעלה אבל ה-user mode יהיה 0 כי מרכיבים קוד של מערכת הפעלה ב-user mode.

- התשובה לשאלת הפופ-אפ של השיעור: "אם ה-CPU יכול להבדיל בין התוכניות שרצות?" לא מדוקנת כי לפי ה-base אפשר לדעת שמרכיצים תכניות שונות.

החלפה בין תהליכיים

מה צריך לשמר מ-process מסוים כדי לעבור לאחר?

נרצה לשמר ערכיים מסוימים למקורה בו נרצה לחזור ל-process הקודם.

נשמר את ה-context של התהיליך אותו רוצים להוציא, תמונה המצב של התהיליך ברגע.

מה שיש ב-context בזיכרון נשאר באותו מקום בו היה ולא מועתק (Instruction, data, heap, stack).

לעומת זאת מה ששמור ב-CPU עומד להזדרס ואת זה צריך להעתיק (stack pointer, PC).

איפה שומרם?

במערכת הפעלה קיים מבנה נתונים בשם **PCB** שיותב ב-kernel memory ומכליל את הערכים של הקונטקסט שצריך לשמר, ומאותחל ע"י מערכת הפעלה כל פעם שמחליפים קונטקסט:

Process ID -

Process state data -

מידע לחישוב עדיפות, מידע על היוזר שמרץ את התהיליך, מידע על מצבים קשורים

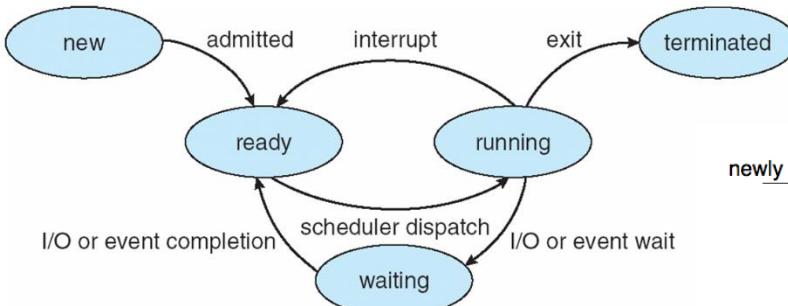
להתקני פלט/קלט -

ערבי הרגיסטרים -

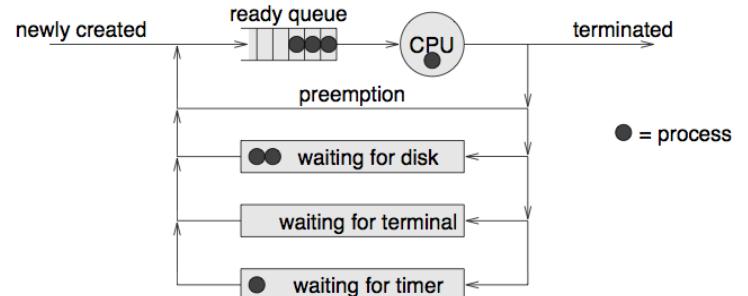
Base, bound -

process state
process number
program counter
registers
memory limits
list of open files
...

מצבים: בשתתלייר נוצר הוא ב-**new** ומucleת הפעלה מתחילה את ה-PCB ← לאחר מכך הוא הופך למצב **ready** ← כאשר הוא ברחידנת הוא הופך למצב של **running** ורץ ב-**CPU** ← אם התתלייר הופסק או מוחכה למשהו בחומרה (דיסק, מקלדת, טיימר וכו') הוא הופך למצב **waiting** ← מתיישחו בסוף התתלייר הסטיים והופך להיות **terminated**.



רוב הזמן התתלייר נמצא במצב הדיאגרמה:

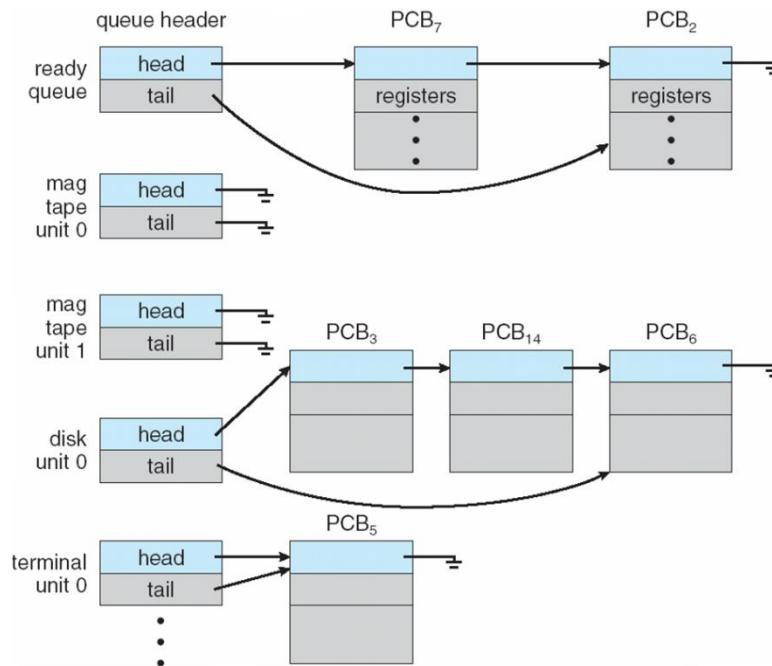


Scheduling

מערכת הפעלה מחליטה על איזה תתלייר רץ ה-CPU ולכמה זמן. היא בוחרת מתוך התהיליבים שמחכים בזמן של ה-ready מייה בNBC של running, כל פעם תתלייר אחד. כמו כן המערכת קובעת ב-time sharing מחליטה כמה זמן התתלייר יהיה ב-ready, running, ואם הזמן עבר היא תחזיר אותו ל-ready.

Scheduler מחליט איזה תתלייר רץ על ה-CPU ולכמה זמן.

Dispatcher מבצע את העתקה של הרגיסטרים, בין היתר ה-**context switch** וחלפת מצבים. לכל מצב יש תורים – רשימה מקוורת של כל התהיליבים (ה-PCB שלהם) שמחכים להיכנס אליו:



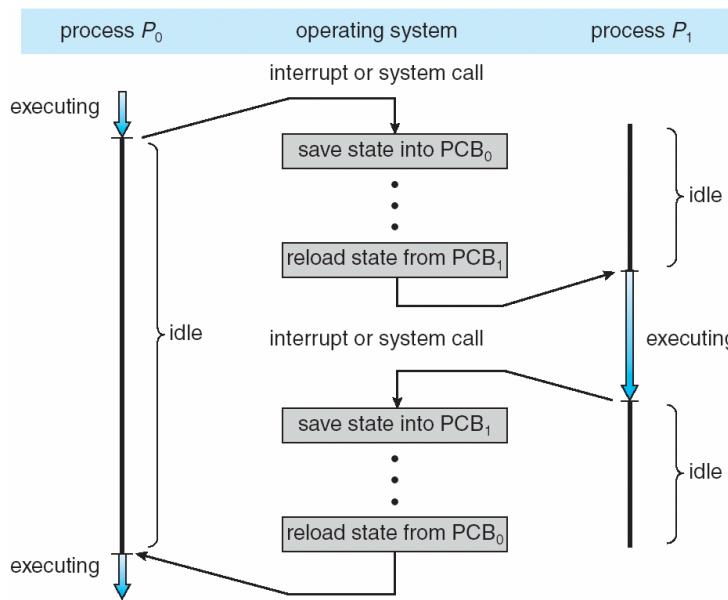
Context switch

החלפת הקשר אחד להקשר אחר.

התהיליך:

- שמירת הקשר של התהיליך, כולל ה-PC והריגיסטרים.
- למצב הנוכחי. PCB עדכן ה-
- הכנסת ה-PCB לתוך הרלוונטי.
- Scheduler בוחר תהיליך אחד לביצוע.
- מעתיק מה-PCB את ערכי הריגיסטר לתוך ה-CPU וمعدכן את ה-PC.
- . user mode kernel mode Restore

דוגמה להרצה:



- התקורה נמצאת בחלקים שבהם לא P_0 ולא P_1 מבוצעים, כי מערכת הפעלה מבצעת את הפעולות שלה.

מתי מחליפים בין תהיליכים?

- פסיקות
 - פסיקת שעון
 - התקני קלט פלט
 - בעיות בזכרון
 - שימוש ב-virtual memory
- System calls
- exceptions

יצירת תהליכיים

קורה במקרים הבאים:

- **אתחול המרבית.**
- **זמן ביצוע הפקודה בשפה עילית - `(fork)`,** בשיווצים תחילך בין התהילך שרך ומפצלים אותו.
- **הרצה תוכנה/פקודה ב-`shell/shell/אפליקציה` יוצרת תחילך אחד או יותר,**
- **הרצה `job`, `batch`,** כמה פעולות אחת אחרי השניה.

```

int pid;
int status = 0;
pid = fork()
if (pid != 0)
{
    /* parent
 */
    ....
}
else
{
    /* child */
    ....
}

```

يُCCR תהליכיים מתאפשר באמצעות תקשורת עם מערכת הפעלה ע"י system programs או ע"י system calls.

דוגמה חשובה: שימוש ב-`(fork)`

בעת הרצת תחילך כאשר יש שימוש ב-`(fork)` התוכנה צריכה להתייחס לשני תהליכיים. מערכת הפעלה יוצרת PCB חדש ושםה את שני ה-PCBים בתורו. ה-`ready`. system call מודיעה על רץ פוק ששייך לאבאו או לבן. אם הערך הוא 0 מבוצע החלק של הבן בקוד, ואם לא איז של האבא, ויצרנו פיצול בתהילין.

הרצאה 4 – Threads Synchronization

תקשורות בין תהליכיים

נרצה למשל שתהליכיים מסוימים יבצעו חלקים ממשימה אחת, וצריכה להיות ביניהם תקשורת לשם כך ועדייןקיימים הפרדה.

Inter-process communication (IPC) – שירות שמערכת הפעלה מספקת כדי לקיים את התקשרות הזו, ע"י calls system специficums עם תקופה גדולה.

קיימות שתי דרכי לבן:

1. **Shared memory system** – הגדרת מקטע משותף בזיכרון לתהליכיים, כך שתהליך אחד יוכל לכתוב לזכרון ותהליך אחר לכתב לו זיכרון וע"י כך להעביר אינפורמציה.

2. **Message passing** – שליחת הודעה מהתהליך אחד אל השני. יש אינטראפיס שמערכת הפעלה מספקת של socket דרכם מעבירים הודעה. ההודעות עוברות דרך kernel וברוכות ב-system calls. יותר בבדה מהדרך השנייה.

Threads

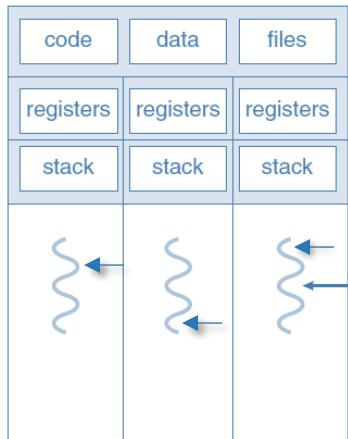
חלק מהתהליכים רוצים לבצע יותר פעולה אחת בו זמן. לדוגמה, הדפסן צריך גם לקרוא מהרשט וגם להציג את הדטה על המסך, ושתי פעולות אלה צריכות להתרחש בו זמן. מעבד תמלילים צריך לעבד את הפעולות על המකלדת תוך הצגת גרפייה על המסך וכו'.

איך מתמודדים עם הצורך ביצוע משימות בו זמן?

1. לא מבצעים את המשימות במקביל, משימה אחת כל פעם.

2. חלוקת תהליך אחד לכמה תהליכיים שבכל אחד מבצע פעולה. יש תלות בין התהליכים שambilאה לצורך בשיתוף מידע – IPC שהוא יחסית איטי ויקר.

פתרונות: Multi-threaded processes

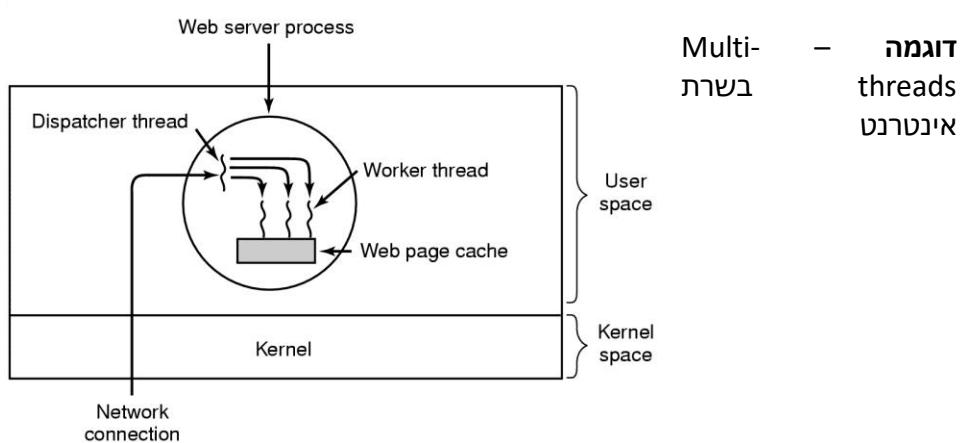


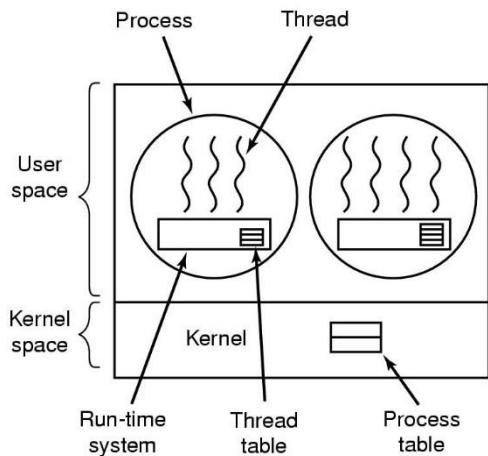
Multi-threaded processes

חלוקת תהליכייםThreads לשניים, "תהליכוניים". נמצאים בתוך התהליכים, לא ישוט עצמאית. Threads שישיכים לאותו תהליך חולקים את הקוד, הדטה וה-heap, אבל לכל thread יש גטיסטרים משלו ו-stack משלו.

בכורה זו לכל thread יש PC משלו, הוא מצביע למקום אחר בזיכרון וביצוע פעולות משלו.

- חשוב לציין שהמקבילות היא עדין בעצם הרצה לシリוגן.



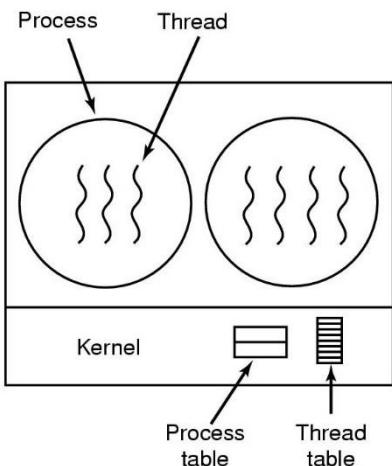
User level threads

- האפליקציה מנהלת את ה-threads עצמה – scheduling – תיעודפים וכו'.

kernel לא מודע לקיום threads .threads מעבר בין threads נעשה ב-user mode ע"י פעולות בתוכנית עצמה.

חרנות:

1. אם מבצע thread system call (מנסה לגשת ל-IO, עושה scheduling) כל התהילך נחסם וכל threads יוצאים מ מצב של running. ככלומר אם קיים thread כלשהו במצב waiting אז כל התהילך יהיה במצב זה.
2. אם התוכןן לקיי, Thread אחד יכול להשלט על השאר.

Kernel level threads

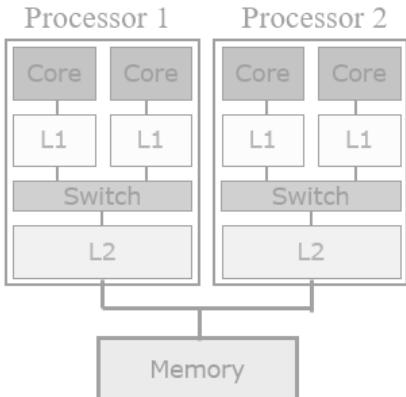
- מערכת הפעלה מנהלת את threads scheduling – ב-PCB יש מצביע ל threads table ומערכת הפעלה עשויה threads שונים מתודדים וקיימים במצבים שונים. kernel threads הם threads שמנוהלים ע"י kernel ורצים ב-kernel kernel level threads הם threads שמנוהלים ע"י kernel ורצים ב-user .user.

לסיכום, היתרונות של רבוי threads לעומתThreads:

1. יצירה, סיום והחלפה של threads הרבה יותר מהיר מאשרThreads.
2. שיתוף הדעתה בין threads באוטו תהיליך זה הרבה יותר זול ונוח מאשר אין הטעבות של kernel .kernel .

- CPU לא מודע ל-threads .threads

<i>processes</i>	<i>kernel threads</i>	<i>user threads</i>
protected from each other, require operating system to communicate	share address space, simple communication, useful for application structuring	
high overhead: all operations require a kernel trap, significant work	medium overhead: operations require a kernel trap, but little work	low overhead: everything is done at user level
independent: if one blocks, this does not affect the others		if a thread blocks the whole process is blocked
can run in parallel on different processors in a multiprocessor		all share the same processor so only one runs at a time
system specific API, programs are not portable		the same thread library may be available on several systems
one size fits all		application-specific thread management is possible



Two processors with two cores and shared memory

Multiprocessors
במציאות יש ריצה של כמה מעבדים במקביל – multiprocessors, ומריצים אפליקציות שונות על מעבדים שונים.
Multi-core – הרצת כמה פקודות במקביל על המעבד. לכל core יש זיכרון שמשל עצמו, ויש זיכרון משותף לライブות שנמצאים קרוב, ועוד זיכרון ראשי של כל הライブות על המעבד.
הרצתה היא מקבילית (ולא לシリוגן). אם יש צורך בתקשורת בין אפליקציות/תהליכיים התקורה אף יותר גדולה.

Hyper-threads

Threads בرمת החומרה, ה-CPU מודע אליהם ויכול לשחלף בלי מעורבות של מערכת הפעלה. (על ליבת אחת)

סינכרונייזציה

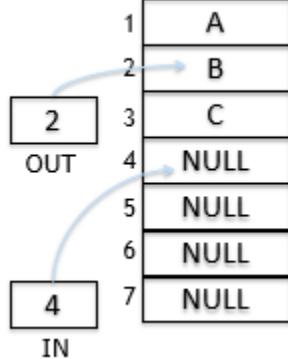
מוטיבציה

זכורת: **spooler** הוא תור שבו נמצאים הג'ובים שראויים להדפס, לפיו ה-printer בוחר איזה ג'וב להדפס והוא משותף לכל התהליכים. נתיחס אליו באל מערך עם שני מצבים:

- IN – המקום הבא שניתן לכתוב ג'וב חדש.
- OUT – הג'וב הבא שציריך להדפס.
- NULL – המקום במערך ריק.

קוד עבור הוספה בקשה ל-spooler (נזכר שהוא משותף כי יכולים יכלים להדפס, ובנוסף גם IN משותף):

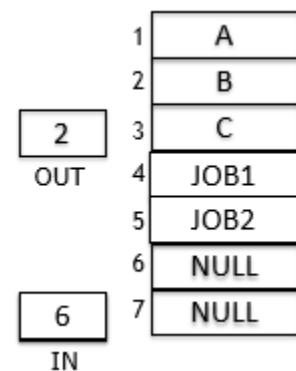
```
Spooler[IN] = Job
IN ++
```



דוגמאות:

הדפסה של שני תהליכים –

```
Process 1: Spooler[IN] = Job1
Process 1: IN ++
<Context switch by OS>
Process 2: Spooler[IN] = Job2
Process 2: IN ++
```

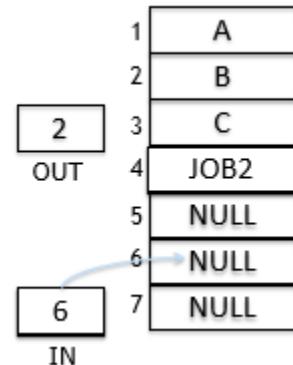


בדוגמה הבאה Job1 נדרס, ואחריו שמדפיסים את Job2 מגעים ל-NULL, כל מה שייכתב אחריו כנראה לא יודפס.

```

Process 1: Spooler[IN] = Job1
<Context switch by OS>
Process 2: Spooler[IN] = Job2
<Context switch by OS>
Process 1: IN ++
<Context switch by OS>
Process 2: IN ++

```



חשוב לשים לב לסייעון בין התהליכים מכיוון ש-IN ו-Spooler משותפים

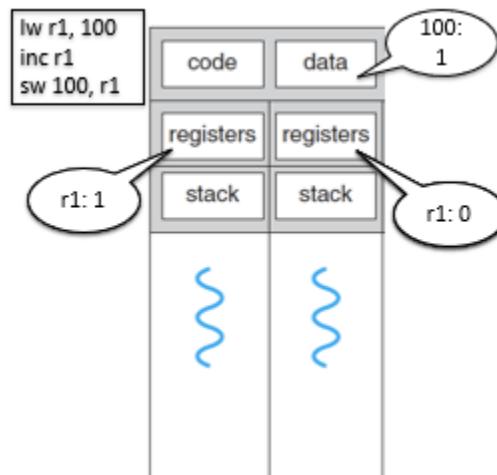
```

IN = 0
Thread 1: lw r1,100
Thread 1: inc r1
Thread 1: sw 100,r1
<Context Switch>
Thread 2: lw r1, 100
Thread 2: inc r1
Thread 2: sw 100, r
    IN = 2
Thread 1: lw r1,100
Thread 1: inc r1
<Context Switch>
Thread 2: lw r1, 100
Thread 2: inc r1
Thread 2: sw 100, r1
<Context Switch>
Thread 1: sw 100,r1
    IN = 1

```

בדוגמה הבאה ההדפסה מתבצעת דרך threads. במקרה זה כאשר מקומפלים ++IN מתקבלות שלוש פקודות אסמבלי (100) זאת הכתובת אליה IN מצביע).

כל thread מעלהレגיסטר שלו את מה שבתו בכתובת 100.

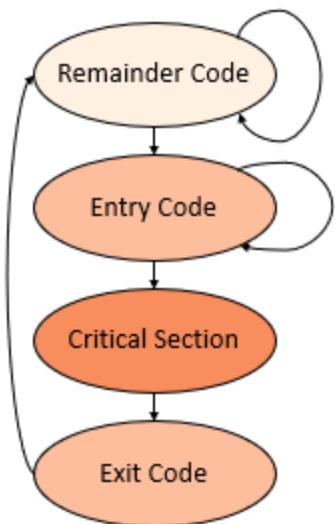


Mutual exclusion problem

הבעיה: גישה (קריאה/כתיבה) לשאב משותף בלי תיאום בין היישויות (תהליכי, threads, ליבות) SMBuzzot זאת. קיימ בקוד "קטע קרייטי" בו מתבצעת הגישה ואילו צריך להתייחס ולתאמ פעולות (בדוגמאות הקודמות ה"קטע הקרייטי" הגישה ל-Spooler ו-IN++)).

נרצה שבל פעם תהליך אחד יבצע את הקטע הקרייטי, ובזמן שהוא מבצע זאת תהליכי אחרים לא יפריעו. כדי לקיים זאת ישנו **אלגוריתם מניעה הדדי mutual exclusion**.

(ישנן בעיות אחרות של סינכרונייזציה בהן נרצה לעשות תיאום אחר ולא מנעה הדדי, אבל זה הדבר הבסיסי שנרצה להבטיח)



חלקי הקוד:

- הקטע הクリיטי critical section
 - השארית remainder – שאר הקוד שהוא לא הקטע הクリיטי.

המערכת לא יכולה לגעת בקטע הクリיטי ובשארית. כדי לקיים את התיאוריה הפעלה "עופפת" את הקטע הクリיטי בקוד נוספת –

 - **Entry code**, בניית הקטע הクリיטי.
 - **Exit code**, אחריו.

נרצה את התכונות הבאות בקטע הקוד כדי לפתור את הבעיה:

- .1 – Mutual exclusion – מניעה הדדית.

2. **התקדיםות Progress** – אם תהליך רוצה להיכנס לקטע הクリיטי איזשהו תהליך מתישחו יוכנס.

3. **מניעת הרעבה Starvation freedom** – אם תהלייך רוצה להיבננס הוא בסופו של דבר ייבננס.

- 4. אם קיימת התקדמות התקנים גם מנייעת הרעה.
 - 5. – **Generality** – התקנים לכל מערכת ולא באופן ספציפי. אין הנחות על המהירות או כמות המשתתפים.
 - 6. – **No blocking in the remainder** – אף תהליך שרעץ מחוץ להעתק הקרייטריון לא יחסום תהליך אחר

פתרונות – 1

בכתבו את ה-**entry code** בר שלא יטפל בפסיכיות, וביציאה מהקווד הקרייטי בחזרה לטפל בהן.

- הפתרון מקיים את תכונות 1,2,3,5 generality לא מתקיים במקרה בו יש כמה מעבדים כי disable interrupt נעשה מ-CPU אחד.

• היזכר לאמר לבצע את זה כי הטיפול בפסיקות נעשה ב-**kernel**, וזה בעיתוי ברגע שיש צורך בפסיקות בקטע הקרייטי. אבל נניח תמיד שהקטע הקרייטי לא נכשל אף פעם, אז הבעיה היא השתלטות ארוכה של תהליכי אחד על המעבד.

Remainder	Remainder
while(turn==1);	while(turn==0);
Critical	Critical
turn=1;	turn=0;
Code for Thread 0	Code for Thread 1

no blocking - progress לא the remainder מתקיימים ונוצר למצוות פתרון אחר.

פתרונות “No cheating”

- קיימשנה גלובלי **turn** שבסימוש רק ב-entry ובי-exit.
 - מניחים שהקטע הクリיטי לא נכשל אף פעמיים.

כל תהליך/thread יש ערך מסוים שכאשר ה-*mutex* בעל הערך זהה נתקע ב-*entry*. בזמן שתת海尔יך נמצא בקטע הקרייטי השאר לא עושים כלום. *Mutual exclusion* אכן מתקיים, הרוכחה:

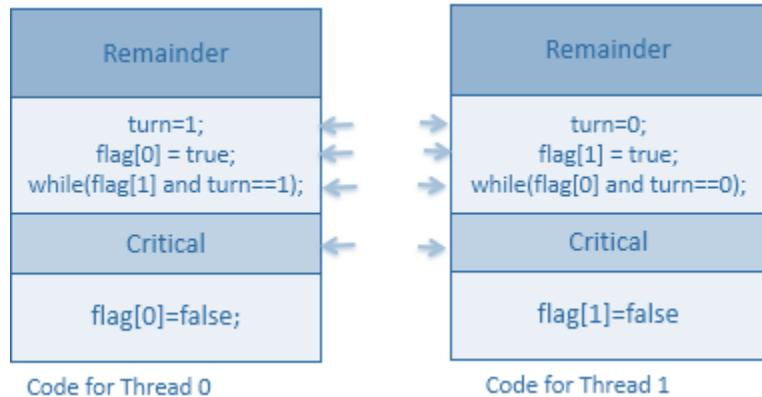
Assume that the two of the processes are in the critical condition, and then w.l.o.g. thread 0 left its entry section (and entered the critical section) first → At that point, turn≠1 → turn=0. The only place turn can be changed to 1 is when thread 0 exits the critical section, thus turn=0 throughout the entire execution of the critical section by Thread 0 → Thread 1 is in its entry code when Thread 0 is the critical section → Contradiction.

הרצאה 5 – Synchronization

Mutual exclusion solutions

שאלת פופ אפ: בהמשך להרצאה הקודמת, נראה דרך נוספת לפתרון "cheating" זו. עבור שני threads ישנו שני משתנים בוליאניים - Thread 0. flag[0], flag[1] מתחילה ב-entry את flag[0] ל-true ומחכה כל עוד flag[1] הוא false. בש-`[1]` יהיה flag[1] true.thread 0 יכול להיבנות לקטוע הקרייטי. Generality לא מתקיים, הקוד ספציפי לשני threads. נשאלת השאלה איזה תכונות בן מתקיימות – mutual exclusion-i-blocking in the remainder-no blocking.

- אם מתקיים היעדר הרעבה ← יש progress. ואם אין progress אין היעדר הרעבה.
 - אם יש progress אין blocking in the remainder-no. אם יש blocking in the remainder-no אין progress.
- הבעיה בפתרון היא שיתכןthreads שchosmis אחד את השני בך שיתכן מצב שבו אף thread לא נכנס.



ביסיון שלישי ל-“cheating” on

שילוב פתרונות 1 ו-2 – יש משתנה turn ומשתנה flag שמסמל האם ה-thread רוצה להיבנות לקטוע הקרייטי. בשיגיע תורו הוא יגיע לקטוע הקרייטי אם ירצה להיבנות.

✗ אין מניעה הדדיות!

Peterson's algorithm

זהה לפתרון הקודם, עם החלפת שורות.

הוכחת mutual exclusion:

נכיה בשילוח שני ה-threads בקטוע הקרייטי, ונניח בה"כ ש-0 thread נכנס ראשון לקטוע הקרייטי. ככלומר יתכן אחד משני המקרים הבאים שגרם לולאת ה-while לסיים: 1. `flag[1] == false`: thread1 לא ביצע את השורה הראשונה שלו בש-0 נכנס לקטוע הקרייטי. ברגע ש-1 הגיע לולאת while הוא לא יצא ממנה ולא יכנס לקטוע הקרייטי כי turn לא משתנה מ-0, בסתיו להנחה בשלילה.

2. `Turn == 0`: Turn == 0 מונע מהthread1 לשוב לולאת while.

✓ No blocking in the remainder – מתקיים באופן כמעט טריוויאלי.

✓ ההוכחה לקיום הרעבה מסובכת ולא תיושה ברגע, וזה גורר progress.

חסרונות:

Code for Thread i

1. מתאים לשני threads בלבד.

2. Busy wait – ביצוע הרבה פעולות של בדיקה וכישלון.

תופעות מרכזיות שגרמו לעיוות

1. **Race condition** – עברו תזמון מסוים מגיעים ל-deadlock ועברו תזמון אחר לא.

2. **Atomic instruction** – פקודות שאין scheduling, context switch באמצעות הביצוע שלhn – קרייה וכתיבה של מילה לזכרון, חיבור מספרים ושמירת התוצאה ברגיסטר וכו'. פקודה לא אוטומטית לדוגמה תהיה `++x`.

.3 – לולאות שרך בודקות תנאי ועושות הרבה פעולות לא חשובות. התהיליכים שזה קורה בהם יהיו ב-ready או running או waiting, ולא ייחסמו. (לולאות שלא עושים כלום)

Remainder
number[i] = 1+max _{j ∈ {1,...,n}} {number[j]};
for j=1 to n { while(i≠j and number[j]>0 and number[j]<number[i]); }
Critical
number[i]=0;

Code for Thread i
(out of n threads)

Lamport's bakery algorithm – אלגוריתם המאפייה

הכללת האלגוריתם של פיטרסון עבור threads א.

כל thread לוקח מספר שבו הוא רוצה להיכנס לקטע הקרייטי, ואיפשהו שמרו המספר של ה-thread שיבול להיכנס. (המחשה במצבת)
 ✕ התנאי שלא יעבד באלגוריתם הוא מניעה הדדית – יכול לקרות מצב שבו שני threads יקבלו את אותו מספר (בגלל התנאי <).

כדי לפתור את זה אפשר להוסיף את הקטע קוד הבא במקום (השינוי מסומן),

```
number[i] = 1+maxj ∈ {1,...,n}{number[j]};  
for j=1 to n {  
    while(i≠j and number[j]>0 and  
          number[j] <= number[i]);  
}
```

אבל במקרה זה יהיה deadlock במקרה ששניים מקבלים את אותו מספר.
דרך נוספת למימוש – סידור לקסיקוגרפי:

```
number[i] = 1+maxj ∈ {1,...,n}{number[j]};  
for j=1 to n {  
    while(i≠j and number[j]>0 and  
          (number[j],j) < (number[i],i));  
}
```

הבעיה – הפקודה max היא לא אוטומית ומחולקת לשולש תת-פקודות: קריית כל הערכים, חישוב המქסימום וכתיבת הערך ב-number.

בסוף דבר, המימוש הנכון יהיה ע"י פירוק הפקודה של השורה הראשונה עם הוספת משתנה choosing = false. במקרה thread.choosing = false. ברגע שסימן – מתקיים FIFO – עד רמה מסוימת שמדובר כל פעם שה-thread בוחר מספר. ברגע שסימן – מתקיים FIFO – עד רמה מסוימת מספר, הוא בודק אם threads לפניו בוחרים מספר. אם כן, הוא יחכה עד שיסיימו ורף אז יבחר.

- התהיליכים נכנסים לפי בסדר.

```
Choosing[i]=true;  
Number[i] = 1+maxj ∈ {1,...,n}{number[j]};  
Choosing[i]=false;  
for j=1 to n {  
    while(Choosing[j]);  
    while(i≠j and number[j]>0 and  
          (number[j],j) < (number[i],i));  
}
```



Read-modify-write instructions

- כיום רוב המעבדים שתומכים גם בפקודות בבדות שמורכבות גם מקריאה וגם מבכיביה, הסוגים העיקריים:
- **Test & set (& lock)** – אם כתובת מסוימת בזיכרון היא 0, כתובים 1 ומחזירים 1. אם היא 1, מחזירים .1

```
{i=*lock; *lock=1; return i;}
```

- דרך להפוך את $++x$ להיות פקודה אוטומית. קוראים איששו ערך, מוסיפים לו inc ומוחזירים אותו.
- **Fetch & add (&p, inc)**

```
{val=*p; *p=val+inc; return val;}
```

- השוואת הערך ל-old, אם שווה כתובים את new. אם שונה .false
- **Compare & swap (&p, old, new)**

```
{if(*p!=old) return false; *p=new; return
```

נראה איך test & set עוזר ל mutual exclusion בפתרון:

שני התהיליכים מנסים לעשות על המשתנה lock את הפעולה test & set בולאלת while. אם אחד מהם יצילח, הוא יקבל את הערך 0 ויבtnob 1, יכנס לקטע הקריטי, וביציא ממנו ישנה את lock החדש ל-0 כך ש-thread יכל לעשות את אותה פעולה. ✓ יש מניעה הדדית.
 ✗ הבעיה – אם starvation freedom so לא מתקיים, אין הבטחה שאם thread אחד יהיה בקטע הקריטי, thread אחר יכנס במקומו.

Remainder
while(test&set(lock));
Critical
lock=0;

Code for Thread i

Burn's algorithm

כדי לקיים .starvation freedom

Remainder
waiting[i]=true; key[i]=1; while(waiting[i] and key[i]) { key[i]=test&set(lock); waiting[i]=false;
Critical
j=i+1 mod n; while(j≠i and not waiting[j]) { j=j+1 mod n ; if(j≠i) { waiting[j]=false; else { lock=0; }

Code for Thread i out of n

Synchronization primitives

פרימיטיב של סינכרונייזציה זה מבנה נתונים אבסטרקטי שמסmorph ע"י מערכת הפעלה, וניתן לשימוש בו לביצוע סינכרונייזציה. יכול להוריד את הצורך ב-wait-busty. מגיע בספריה שממומשת ע"י מערכת הפעלה. הפעולות הן אוטומיות ולא יהיה context switch באמצע.

Semaphore

מחלקה עם שני שדות פנימיים – ערך ורשימה של תהליכי threads.

שלוש מethodות:

1. אתחול

Init(S,v)

S.value = v

2. הורדת ערך – הורדת הערך ב-1 בצורה אוטומטית. אם הוא שלילי, מוסיפים את ה-thread שקרה לפונקציה לרשימה ומורדים אותו (מעבירים אותו למצב waiting לחמן בלתי מוגבל).

Down(S)

```
S.value = S.value - 1
if S.value < 0 then
{ add this thread to S.L;
sleep();}
```

3. העלאת ערך – מעלים את הערך ב-1. אם הערך עדין שלילי, לוקחים תחיליך מהרשימה ומעבירים אותו למצב ready.

Up(S)

```
S.value = S.value + 1
if S.value ≤ 0 then
{remove a thread T from S.L;
Wakeup(T);}
```

Remainder	
down(lock)	
Critical	
up(lock)	

Code for Thread i

דוגמה למניעת הדדיות עם שימוש ב-semaphore :

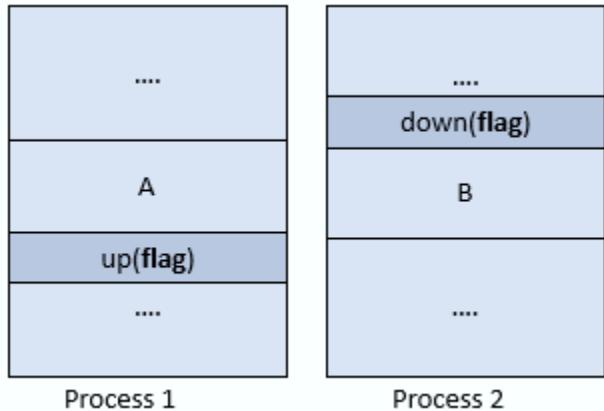
יש semaphore אחד בשם lock, משותף לכל התהליכים ומאותחל לערך 1 – לכל היותר thread אחד יוכל להיכנס לקטע הקרייטי. כל thread שרצה להיכנס לקטע הקרייטי יעשה לפניו down. אם יש כבר תחיליך בקטע הקרייטי הערך יփוך לשילי וה-thread יעבור למצב sleep.

ביציאה מחזרים את הערך ל-1 וה-thread הבא יוכל להיכנס.

תכונות שמתקיימות:

- ✓ Mutual exclusion
- ✓ Progress
- ✓ No blocking in the remainder

✓ רק אם הרשימה תהיה FIFO.
Semaphore – **Binary semaphore**

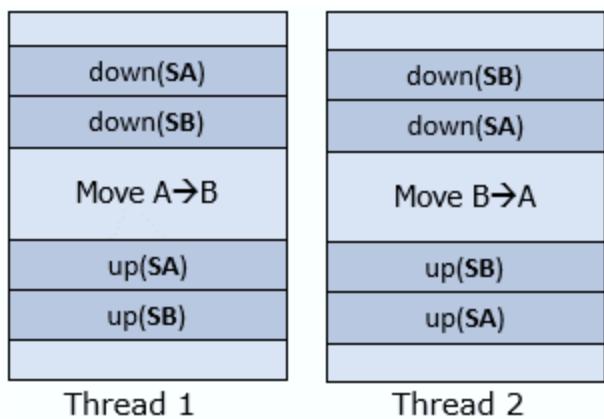


Execute A after B

בעה נוספת שה-semaphore פותר.

למשל יש שני תהליכים שMRI'רים קודמים שונים ונרצה שקטע מסוים בתהיליך אחד (B) יתבצע רק אחרי קטע מסוים בתהיליך השני (A). אין קטע קריטי.

יש איזשהו semaphore שנקרא flag ומאותחל ל-0. B לא יתבצע כל עוד ה-flag לא יאותחל ל-1. התהיליך ה-1 יעלה את הערך ו"יעיר" את התהיליך השני ו-B יוכל להתבצע.



Semaphore and silver bullets

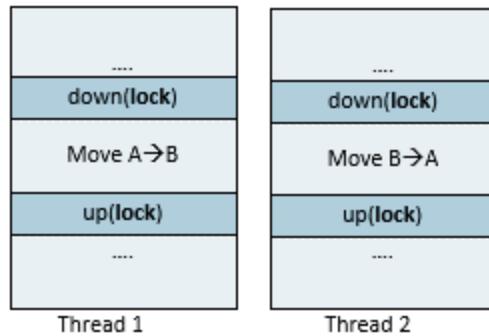
semaphore לא פותר את כל הבעיה, דוגמה אפשרית היא העברת בספ' מחשבון לחשבון. לא נרצה שבזמן ההעברה תהיה גישה לאוטו חשבון. המשך בשבוע הבא.

הרצאה 6 – sync cont

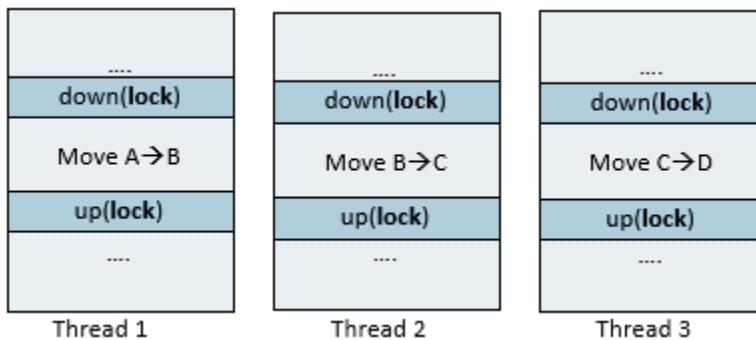
Moving money between accounts problem

הבעיה: יש שני threads, כל אחד רוצה להעביר כסף מהחשבון של השני.

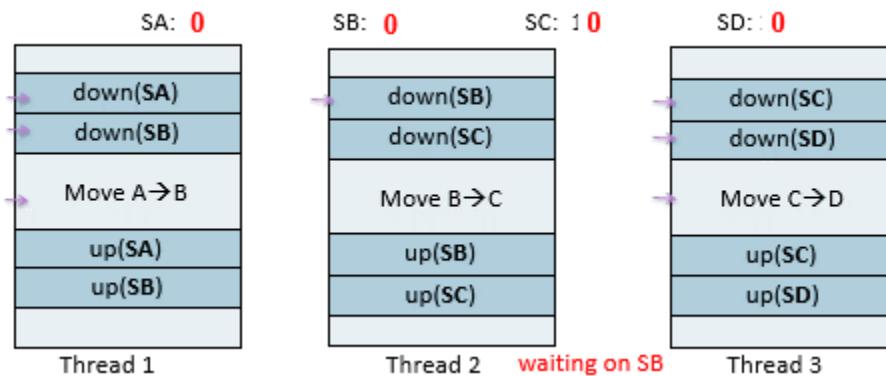
הפתרון הרגיל – חסימת הקטע הקריטי של ההעברה ע"י מימוש מניעה הדדית עם מוטקס (הערך של הסמפור הוא 1). Thread 1 עושה down lock-ל-Thread 2 כדי להיכנס לקטע הקריטי. בשזהו מצליח הוא נכנס לחשבון מעביר את הכסף מ-A-ל-B, וכשמסיים עושים עבורה up כדי שהthread השני יוכל לצאת להיכנס.



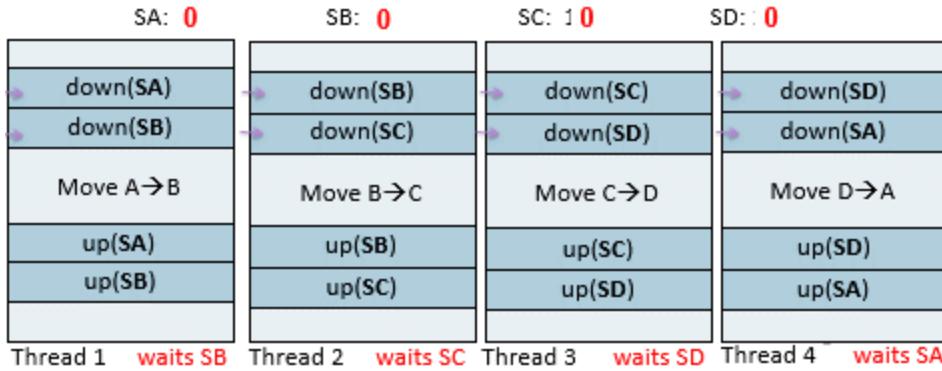
במקרה שבו יש שלושה threads, אותו פתרון גם יעבוד אבל יהיה לאiesel, כי שני threads מתוכם בן יכולים לעבוד במקביל (למשל 1 ו-3 שעורכים את ההעברות מחשבונות אחרים).



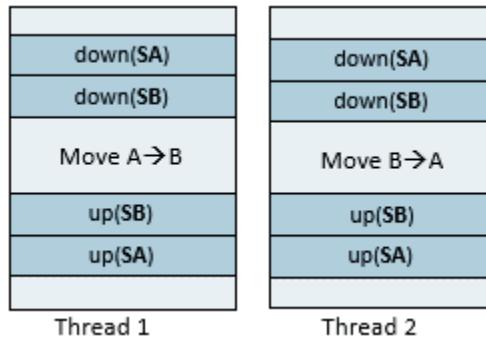
הפתרון: שימוש ב-lock לכל חשבון כך שבכל העברה חוסמים



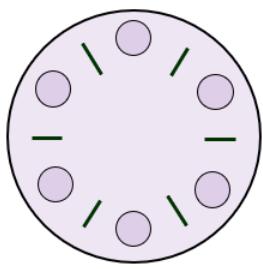
הבעיה: במקרה בו יש ארבעה סטפרים המתחילים ל-1, אם כל thread יבצע את השורה הראשונה, יהיה deadlock.



הפתרון: לעיתים כישיש לנו הרבה משאבים ונרצה לנעול אותם אחד לאחר השני בסדר (למשל במקרה של שני חשבונות, קודם להוריד את SA ורק אחר כך את SB).



בעיית הפילוסופים הסודדים

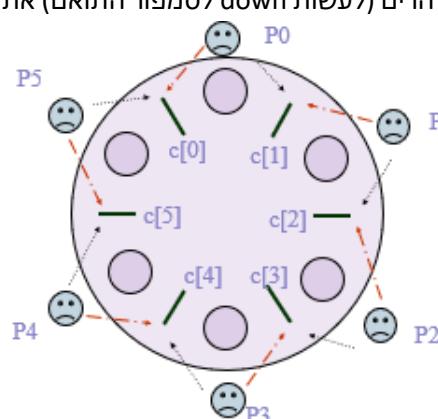


הבעיה: פילוסופים יושבים ב圍ג'ל וועשווים שני דברים: חושבים ואוכלים. האכילה נעשית ע"י שני צ'ופסטיקים, לכל פילוסוף יש צ'ופסטיק מימין וצ'ופסטיק משמאלי. ברגע שפילוסוף סיטם לאכול הוא מוריד את הצ'ופסטיק ומתחילה לחסוב עד שניה רעב שוב וחוזר על התהילה מחדש. ← מספר הצ'ופסטיקים הוא כמספר הפילוסופים. הפילוסוף הראשון שלקח את המקל זוכה לאכול אותו והפילוסוף השני צריך לחכות שהוא יסיים, במידה שהוא שירצה לאכול.

פתרון 1: כל פילוסוף הוא תהליך. הצ'ופסטיקים הם כמו משאב ויש סטפר לכל צ'ופסטיק שמתחיל ל-1 כי רק פילוסוף אחד יכול להחזיק בו. כל פילוסוף ינסה להרים (לעשות down לסטפר התואם) את הצ'ופסטיק לשמאלו ואז את הצ'ופסטיק לימינו.

down(chopstick[(i+1)mode 6])
down(chopstick[(i)])
Eat
up(chopstick[(i+1)mode 6])
up(chopstick[(i)])
Think

Code for philosopher i



הבעיה: deadlock, אם כל אחד ינסה להרים את הצ'ופסטיק השמאלי, כאשר הוא יעבור לימני הוא ייתקע בofilosof אחר בבר הרים אותו. (אדום מהבהה, שחור ללחץ)

הפתרון: שבירת סימטריה – פילוסופים מסויימים (0-4) ירימו קודם את הצ'ופסטייק השמאלי ואז את הימני, והפילוסופים האחרים (5) יעשו להפוך.

כדי להוכיח שהפתרון עובד צריך להראות שמתקיים progress, שאם פילוסוף מנסה להרים צ'ופסטייק בסופו של דבר הוא או מישחו אחר יצליה. נניח שפילוסוף מנסה להרים את אחד הצ'ופסטייקים ולא מצליח, אבל אף פילוסוף אחר לא אוכל.

נחלק למקיריים:

1. **פילוסופים 0, 1, 2, 3 ניסו להרים את הצ'ופסטייק השמאלי.** נראה שאנו שham הצליחו, או שלא ומישחו אחר יצליה.

נחלק לשני מקיריים:

- **פילוסוף 3 ≤ i ≤ 0 ניסה להרים את הצ'ופסטייק השמאלי 1 + i ולא הצליח** ← זה אומר שני שלידי הרים אותו, בلومר פילוסוף 1 + i. אותו פילוסוף בעצםלקח את הצ'ופסטייק הימני לו, ומכאן אפשר להסביר שהוא בבר לתקח את השמאלי ← יש לו את שני הצ'ופסטייקים והוא יכול להתחיל לאכול – סטירה.
- **פילוסוף 3 ≤ i ≤ 0 בבר לתקח את הצ'ופסטייק השמאלי 1 + i ← עבשו הוא** מנסה לחת את הימני i כדי להתחיל לאכול ← אם הוא לא מצליח זה אומר שהפילוסוף לידיו 1 - i לתקח אותו ← אותו פילוסוף שלקח אותו צריך לחת גם את הצ'ופסטייק שלימינו 1 - i ← אם הוא לא מצליח עוברים הלאה עוד ועוד. בסוף מובטח שייה פילוסוף שייקח את הימני כי אם נגיע לפילוסוף ה-0 שלא יכול לחת את הצ'ופסטייק לימינו, זה אומר ש-5 לתקח אותו. אבל בשביל 5 זהו הצ'ופסטייק השמאלי, בلومר הוא בבר לתקח את הימני והוא יכול להתחיל לאכול – סטירה.

2. **פילוסוף 4 ניסה לחת את צ'ופסטייק 5 (לשמאלו) ולא הצליח** ← 5 ניסה לחת אותו. בשביל 5 זה הצ'ופסטייק הראשון (הימני) והוא מחייב לצ'ופסטייק השמאלי 0 ← אם הוא באמת מחייב זה אומר ש-0 לתקח אותו ואת המקרה הזה הוכחנו מוקדם. אם לא אז הוא הצליח ואכול – סטירה.

3. **פילוסוף 5 מנסה לחת את צ'ופסטייק 0 (לשמאלו) ולא מצליח** ← 0 לתקח אותו שבשבילו הוא הצ'ופסטייק השני אז הוא יאכל. **אם 5 מנסה לחת את צ'ופסטייק 5 ולא הצליח** ← 0 מחייב ש-4 לתקח אותו, ו-4 מחייב לימני (צ'ופסטייק 4). אם הוא לא הצליח אז 3 לתקח אותו ומחייב לצ'ופסטייק 3, וכן הלאה וכן הלאה. בסוף נגיע לכך שפילוסוף 0 לתקח את הצ'ופסטייק השמאלי ומחייב לימני 0. אם הוא לא מצליח לחת אותו אז 5 בבר לתקח אותו, אבל 5 לתקח קודם 5 ואז את 0, בلومר הוא בבר לתקח את 5 – סטירה.

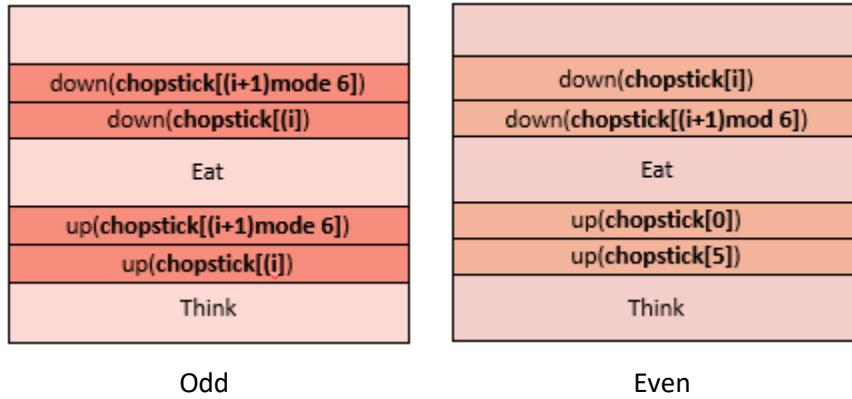
תלו אייר הסמפור ממומש.

תנאים ל-deadlock

- כל התנאים צריכים להתקיים במקביל.
- 1. **מניעה הדידית** – נעלמת משאבים.
- 2. **יותר משאב אחד** – בר שיש מצב ש-thread נעל משאב אחד אבל עדין מחייב למשאב الآخر.
- 3. **Non-primitive allocation** – אם thread נעל משאב מסוים רק הוא יכול לשחרר אותו.
- 4. **מעגליות**.

ניתן להסתכל על הבעיה בעל גרעף עם קודקודים – קבוצה אחת של קדקודים היא של threads/פילוסופים, ובköוצה שנייה היא של המשאים/צ'ופסטייקם. יש צלעות בין משאב לבין thread – צלע יוצאת מהמשאב ל-thread אם הוא מחזיק במשאב (עשה chowd והצליח), צלע יוצאת מה-thread למשאב אם thread מחזק. אם יש מעגל אז יש deadlock.

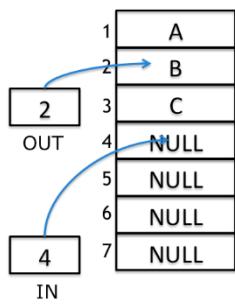
פתרון נוספת: נשבור את הסימטריות ע"י כך שהפילוסופים הזוגיים ייקחו קודם את צ'ופסטיק שמאל ואז את ימין, והאי זוגיים להפך. הפתרון יותר יעיל שכן תמיד שני פילוסופים יוכלו לאכול יחד.



- מערכת הפעלה לא מסוגלת להתחמודד עם deadlock.
- הבעיה עם deadlock היא שהיא לא תמיד קורה, הוא תלוי בתזמון וב-context switch.
- הדרך הטובה להתחמודד עם זה (למנוע מעגליות) היא למספר את המশאים, לנעל מההנמרק גבוה ולשחרר מהגבוה לנמוך.

שאלת מבחן - הסמפור בבעיה מוצחן ל-6. יכול להיות deadlock כי אם כל פילוסוף מורד את הסמפור אפשר להיתקע.

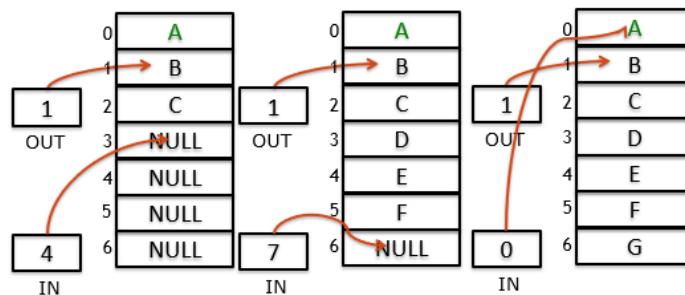
בעיית producer-consumer



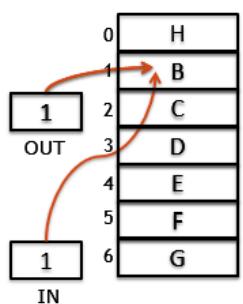
קובץ אחת של threads מיצרת עבודה וקובצת אחריה צריכה לעבודה. לדוגמה – במקורה של spooler שבו מכניםidosים לעבודות לתור של המדפסת, ה-threads שמעדכנים את הבאפר היו היצרנים והמדפסת היא הצרכן.

באפר ציקלי

麥iouן שהבאפר הוא סופי נשתרמש בו בצורה מעגלית – **באפר ציקלי** (בහנחה שמי שנכתב במקום בבר נוצר).



במקרה שבו ה-IN וה-OUT שוים, איך נדע אם הבאפר מלא או ריק?



פתרון 1: לשמור איזשהו counter שייהווה אידיקציה למקומות התפוסים בבאפר. כשהיצרך משתמש הוא מעלה את ה-counter ושהצרכן משתמש מעתה הוא מורד. אצל היצרך יש לולאת while שגורמת לו לחכות עד עוד הבאפר מלא, ואצל הצרכן לולאה שמחכה כל עוד הבאפר ריק.

- הפקודות צריכות להיות אוטומיות (fetch&add) / בקטע קרייטי עטוף במניעה הזדונית.

Producer

```
while (COUNT==n);
buffer [IN]=job;
IN=IN+1 mod n;
COUNT++;
```

Consumer

```
while (COUNT==0);
job=buffer [OUT];
OUT=OUT+1 mod n;
COUNT--;
```

פתרון 2: וויתר על מקום אחד במאפר, ובדיקה האם IN מצביע מקום אחד לפני OUT. אם כן, נתיחס לבאפר בעל מלא. הפתרון הזה טוב רק בשחזרן והיצرنם ייחדים.

Producer

```
while ((IN+1 mod n)==OUT);
buffer [IN]=job;
IN=IN+1 mod n;
```

Consumer

```
while (IN==OUT);
job=buffer [OUT];
OUT=OUT+1 mod n;
```

פתרון 3: טוב לבעה הכללית – כמה צרכנים וכמה יצרכנים. יש שלושה סטפוריום:

1. **מוחקם** – מואתחל ל-1. נרצה להגן על עדכון משתנה ה-IN ומשתנה ה-OUT. רק thread אחד יכול להיכנס

לקטע שבו הוא מעדכן אותו ולשנות אותו כל פעם.

2. **Empty** – מייצג את התוור של היצרכנים, סופר את מספר התאים הריקים.

3. **Full** – מייצג את התוור של הצרכנים, סופר את מספר התאים המלאים.

Producer

```
down(empty)
down(mutex)
buffer [IN]=job;
IN=IN+1 mod n;
COUNT++;
up(mutex)
up(full)
```

Consumer

```
down(full)
down(mutex)
job=buffer [OUT];
OUT=OUT+1 mod n;
COUNT--;
up(mutex)
up(empty)
```

Initial values

```
mutex = 1
empty = n
full = 0
```

• היצרן מוריד את מספר התאים הריקים ב-1 לפני שנכנס לקטע קוד, וכמשמעותו מעלה את מספר התאים המלאים

ב-1. היצרך להפנ. כך שאם התוור ריק היצרכנים יחכו לתורם עד שיתמלא, ואם התוור מלא היצרכנים יחכו.

Monitors

מבנה נתונים שמצויר מבינה שלו אובייקט וככל:

- מבני נתונים מסוותפים.

- פרוצדורות, מתודות שניתן לביצוע על הדטה.

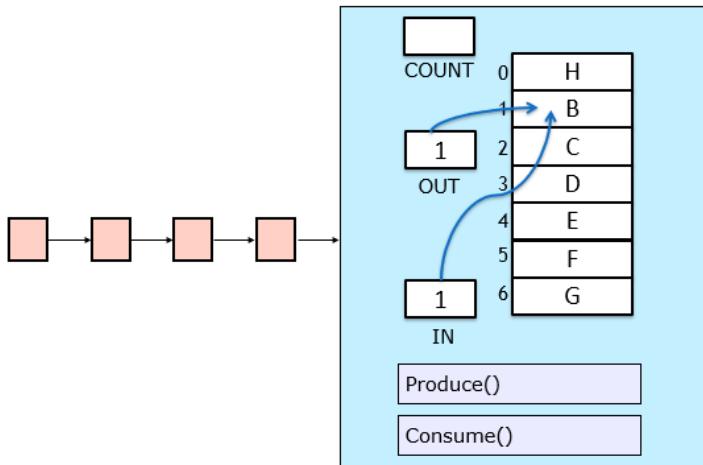
- סינכרונייזציה בין התהליכים שמנסים לגשת לדאטה ולבצע את הפרוצדורות.

ניתן לגשת לדאטה ורק מטור המוניטור.

המוניטור מספק מעין מכונה הדדית אוטומטית, וברגע ש-thread ינסה לבצע פעולה על הדאטה בזמן ש-thread אחר מבצע, הוא ייחסם (די מגביל, אבל קל לשימוש).

הרצאה 7 – Sync, Memory Hierarchy and Caching

Monitors



שימוש במונייטורים בעקבית

הדעתה המשותף יהיה הבאר, המצביעים, וה-counter.

שתי פעולות שניתן לעשות על הדעתה –

- `Produce()` – הוספה עבודה לבאר.

- `Consume()` – קראת עבודה מהבאר.

בר בעצם thread יכול או לצורך או ליצור ולא יהיה מצב שני threads ניגשים לבאר.

הבעיה: נניח שצרנו ניגש לבאר ריק, הוא הרוי ייחכה במוניטור עד שהבאר מתמלא. אבל לא ניתן למלא את הbeer ב-*the monitor* עד תפוס, וכבה יתתקע.

Condition variables

ונundo למנוע את המקרה הקודם.

בעצם שימושו במצב של "waiting", הוא "ichba" באיזושהי נקודה עד שהוא ייחכה יחרר אותו וישלח לו סימן לכך דרכו המשתנים שהם חלק מהמונייטור.

ישן שלוש פקודות שניתן לבצע על המשתנים:

1. **Wait** – שחרור הנעה של המוניטור בר-*thread* אחר יוכל להיכנס. ה-*thread* שנמצא עכשו במונייטור ייכה לסיגナル.

2. **Signal** – מעיר *thread* שנכנס להמתנה (למשל אם יש צורך שמחכה כי הbeer ריק, יצרן נמצא בעת במונייטור ישלח את הסיגナル ברגע שישים לכתב את העבודה שלו בbeer, אך באותו זמן יוכל להשתמש עכשו בbeer). (notify בג'אווה)

3. **Broadcast** – מעיר את כל ה-*thread* שנמצאים בהמתנה. (notifyAll בג'אווה) (דוגמה לשימוש במצבת)

פסודו קוד בג'אווה:

```
public synchronized void produce(char ch)
{
    while (COUNT == BufferSize)
        wait();
    store[IN] = ch;
    IN=(IN+1)%BufferSize;
    COUNT++;
    notifyAll();
}
```

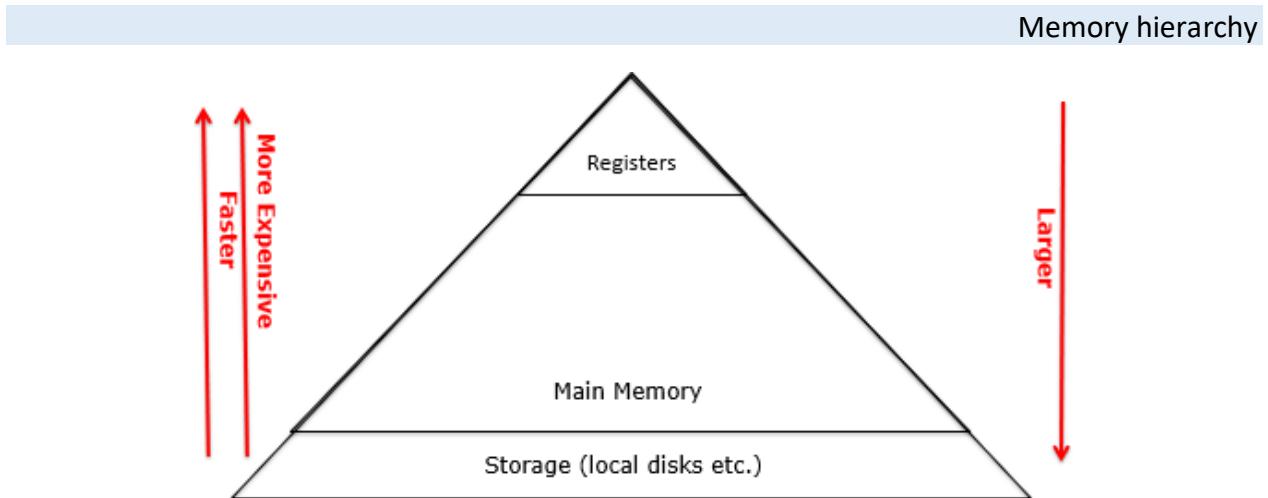
```
public synchronized char consume()
{
    while (COUNT== 0)
        wait();
    ch = store[OUT];
    OUT=(OUT+1)%BufferSize;
    COUNT--;
    notifyAll();
    return ch;
}
```

- `wait` נעשה אחרי איזושהי ולאת תנאי (ולא תנאי יחיד) למקורה שבו שחררנו בביטחון מה תהליכי (עם `notifyall`) כדי לאפשר לתהליך אחד בלבד להיכנס.
- מעדיף לעשות `All notify` כדי לא לפספס סיגנלים.

החלפת אינפורמציה בין תהליכיים

לפעמים המשאב המשותף נדרש להחליף אינפורמציה בין ה-threads/תהליכים, וזה יכול להתבצע בשני מודלים:

1. **Shared memory** - מקום משותף בזיכרון שניין לבולם לגשת אליו.
2. **Message passing** - העברת הודעות.



RAM

בכונן Registers ו/או בזיכרון הראשי, הגיעו לכל הכתובות נעשית באותו מחיר בלי קשר למיקום שלהן.

RAM סטטי (SRAM) – יותר מהיר וייתר יקר, פחות רגיש לרעש. המידע נשמר כל עוד המבשיר דלוק.

RAM דינמי (DRAM) – זול, איטי, יותר גדול, צריך ריענון כל 10 עד 100 מיליאניות, רגיש לרעש.

- צריך הרבה מהזיכרון הראשי וכן יאוחסן ב-DRAM.
- גטיסטרים יוממשו ב-SRAM.

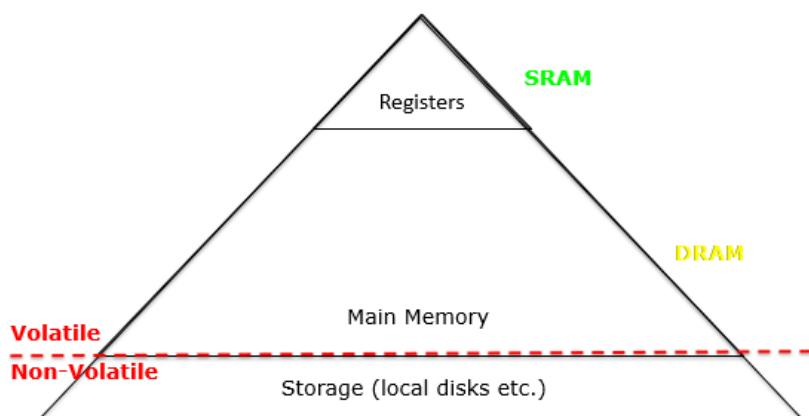
אם ברגע שמכבים את המחשב הדטה בזיכרון נעלם, הוא נחשב **זיכרון נדייף**. (למשל RAM, DRAM, CD-RW). אם נשמר, הזיכרון **לא נדייף**.

Disk

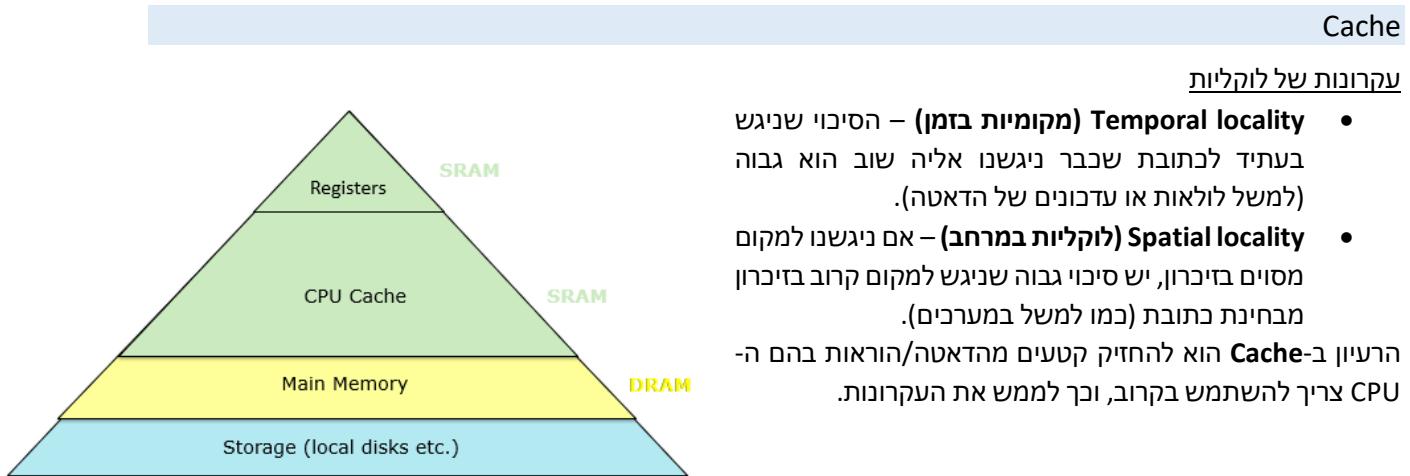
הdisk נמצא ב-storage. המהירות בתובנה ביחסות של סיבוב בדקה. הדטה כתוב על פני הדיסק והזרוע קוראת אותה ממנו.

- גישה לדיסק היא איטית יותר.

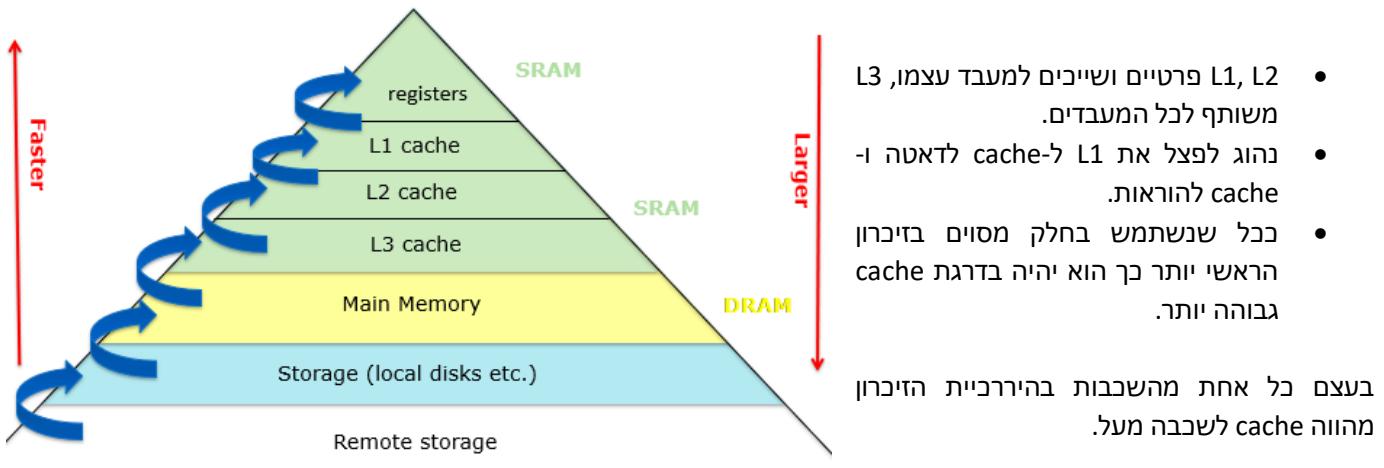
• גם ב-storage יש היררכיות פנימיות.



ROM – זיכרון שהכתיבה אליו איטית וקשה ולכן בד"כ הוא משמש לשמרות קושחה והפקודות הראשונות של המחשב כשהוא עולה. לא חלק מערכת הפעלה.



בעצם מכך שיש במעבדים יש במא רמות של CPU cache:
 $L1 \leftarrow L2 \leftarrow L3$



	Server	Mobile device
Registers	0.3 ns / 1 KB	0.5 ns / 0.5 KB
L1	1 ns / 64 KB	2 ns / 64 KB
L2	3-10 ns / 256 KB	10-20 ns / 256 KB
L3	10-20 ns / 2-4 MB	None
Main Memory	50-100 ns / 4-16 GB	50-100 ns / 256-512 MB
Storage	Disk: 5-10 ms / 4-16 TB	Flash: 25-50 ms / 4-8 GB

טרמינולוגיה

Cache hit – מנסים למצואו איזשהו פרט ב-cache והוא אכן שם. היחס בין מספר הפעמים שפגענו לבין מספר הפעמים שניגשנו.

$$\text{Hit rate} = \frac{\# \text{hits}}{\# \text{access}}$$

Cache miss – מנסים למצואו איזשהו פרט ב-cache והוא לא שם. אם זה קרה, נדרש לדוח בהיררכיית הזיכרון כדי למצוא אותו.

$$\text{Miss rate} = \frac{\# \text{misses}}{\# \text{access}}$$

הזמן שלוקח למצוא את אותו פרט בשלבים נמוכים יותר בהיררכיה.

דוגמאות:

- נכיח ש-hit הוא 100%, הזמן שייקח לעשות חיפוש הוא 100 גישות (100 גישות, כל אחת ב-1 ננו-שניה).
- אם ה-hit הוא 90% אז נעשה חיפוש של 100 גישות כל אחת 1 ננו-שניה, אבל את ה-10% הננותרים נדרש 10 גישות לפחות 10 ננו-שניות ב-2, סה"כ 200 ננו-שניות (עליה של 50% בזמן החיפוש).
- במציאות, ה-hit ratio של 1L הוא 97%-95%

דילמות בשימוש ב-cache?

- אויר למסמך גישה מהירה?
- מה כדאי לשמר ב-cache?
- מה כדאי להוציא מה-cache כשהוא מלא?
- מה קורה אם כותבים לאייר ב-cache?

מימוש גישה מהירה - Direct mapping

בתובת בזיכרון הראשי היא בגודל 32 ביט או 64 ביט, לעומת cache שהוא יוטר קטן. יש צורך למפות בתובת בזיכרון לכתובת-cache כך שהחיפוש יהיה מהיר.

לדוגמה: הזיכרון הוא בגודל 13 ביט (2^{13} מילימ) וה-cache בגודל 2 ביט (4 מילימ).

נרצה שמה שבכתובת **1011111111111** יהיה כתוב גם ב-cache. כדי להחליט איפה לאחסן אותה, נפצל את המילה ונסתכל על שני הביטים הראשונים – **10**. נלק ל-cache וניגש לכתובת **10**, ונרצה שמה יימצא בכתובת זו יהיה באותה כתובת מתאימה בזיכרון.

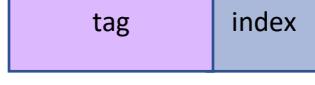
הבעיה: יש מילימ נוספים (כתובות נוספות בזיכרון) שמתחלילות ב-10. אם נפצל אותן וניגש ל-cache, נדרס את הערכיהם.

הפתרון: לשומר ייחד עם מה שבכתוב בזיכרון **tag** – החלק השני של המילה (11 הביטים הנונתרים במקורה שלנו – **1111111111111**). אם בשמחפשים את המילה בכתובת **1011111111111** ב-cache מופיע התג המתאים, זה cache hit.

פתרונות עובד, אבל שומרים עוד אקסטה ביטים רק כדי לדעת אם זה נכון או לא וזה מעלה את התקורתה.

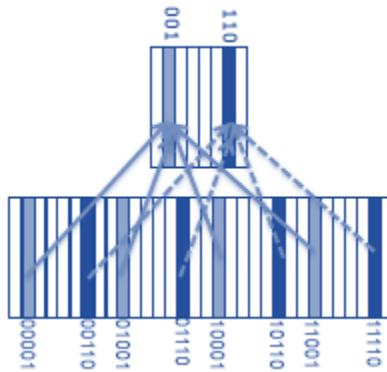
← כל כתובת בזיכרון מזופה למקום אחד ייחיד!

← שתי כתובות שונות שיש להן אותו אינדקס לא יכולות להיות ממופות בו זמינות!

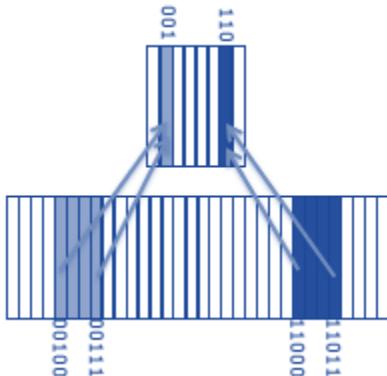


בגל עיקון הלוקליות במקומות עדיף לנו למפות מילה באופן הבא:
(בלומר האינדקס הוא מימין המילה)

בדרכ ה затה נוכל לשמר ב-cache בתובות קרובות בזיכרון, כי הן ימopo למקומות שונים ב-cache, הרבה פחות miss cache:



לעומת ה דרך השניה (האינדקס הוא שמאלי למילה), שבתובות קרובות ימopo לאוטו מקום(cache), ומכיוון שבתובות שונות לא יכולות להיות ממופות לאוטו מקום בו זמינות זה יהיה בעית:



נרצה להוריד את התקורה של התג: בששומרים cache, שומרים בлокים ולא מילים כאשר כל בлок מכיל 8 מילים. הפעולות הן על בלוקים ומיצירות התג פר בлок וזה מביא להורדת התקורה פי 8. בשוחצים לבLOB מילה, כתובים בлок. בשוחצים לפנות מילה, מפנים בлок. לעיתים נאחסן דברים מיוחדים. מיפוי ה כתובות:



ה-offset מצביע על איזו מילה בבלוק נרצה.

דוגמא:

- הדיבורן: 256 מילים, כל מילה 8 ביטים.
- cache: 32 מילים, מחולק לשני סטים.
- כל בлок הוא בגודל 8 ביטים, מכיל 8 מילים.
- 2 ביטים לייצוג הבלוק, 3 ביטים לייצוג offset. סה"כ גודל התג יהיה $3=2-3-8$

← איך ניתן לכתובת 178? (10110010)

צריך לפרק את ה כתובות ל-3,2,3. נקבל:

Offset = **010** block = **10** tag = **101**

אם אכן התג המתאים באותו בлок הוא **101** אז יש hit cache, ונחזיר את התוכן במקום **10010**.

הרצאה 8 – Cache

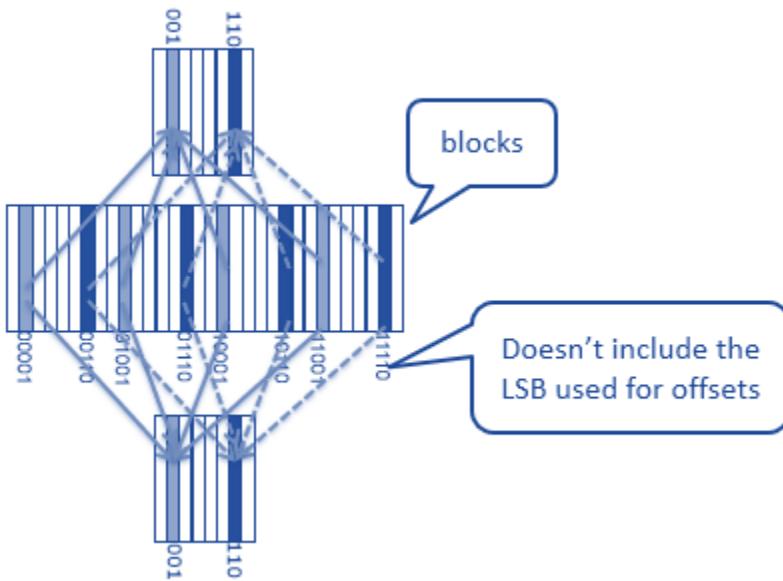
Addresses Translation

~ היסכום מבוסס על ההקלות של הרצאה משנה שעבירה ~

Set Associativity

בשיטת מיפוי היישר כתובות שמופות לאותו מקום לא ימצאו בו זמנית ב-cache. בשיטתה-set associativity, cache מוכפל מחולק לסטים (כל סט נקרא way) ובכל בלוק יכול להימצא בכל אחד מהסטים. שתי מילימ שימפו לאותו מקום בזיכרון אין יכולות להיות בו באותו רגע נתון, כל אחת בסט אחר.

- בשמחפשים מילה מהחפשים בשני הסטים. אם היא נמצאת באחד מהם יש hit.cache.
- פחות קונפליקטים.
- נעשה במקביליות בחומרה.



דוגמה:

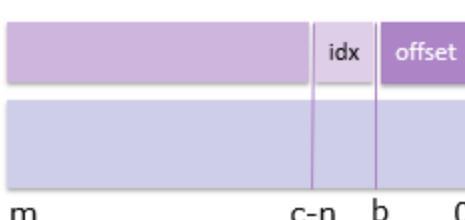
- איך נמפה את כתובת 178 (10110010)?
- הזיכרון: 256 מילימ, כל מילה 8 ביטים.
 - cache: 32 במילימ, מחלק.
 - בלוק הוא בגודל 8 ביטים.
 - לכל way יש שני בלוקים ולכן צריך ביט אחד של אינדקס.
 - 3 ביטים לייצוג ה-offset.
 - לייצוג התג נדרש 4-3-1=8 ביטים.
 - Offset = 010 Index = 0 Tag = 1011

אם הבלוק באינדקס 00 או באינדקס 01 הוא עם-tag 10, יש hit cache ווחזר את הערך בлокם 1010 או 1000 אחרית יש cache miss.

n-way associativity – יש n אפשרויות caches, n .

Fully associativity – כל המיקומים אפשריים, כל cache הוא בגודל 1 ובודקים בכל הבלוקים במקביל.

- האסוציאטיביות תליה בתמיכת החומרה.
- בכל שאסוציאטיביות יותר גבוהה יש פחות קונפליקטים.



חישוב index, offset, tag
הזיכרון: 2^m ביטים
Cache: 2^c ביטים
בלוק: 2^b ביטים
עבור בתבנת X:
 $X \bmod 2^b - \text{Offset}$
 $(X \div 2^b) \bmod 2^{c-n-b} - \text{אינדקס}$
 $X \div 2^{m-c-n} - \text{טג}$

-
-
-

כל cache ממומש בדרך שונה:

Characteristic	L1	L2	L3
Size	32 KB I/32 KB D	256 KB	2 MB per core
Associativity	4-way I/8-way D	8-way	16-way
Access latency	4 cycles, pipelined	10 cycles	35 cycles
Replacement scheme	Pseudo-LRU	Pseudo-LRU	Pseudo-LRU but with an ordered selection algorithm

מקורות ל-cache misses

1. **Compulsory** – בשניגשים פעם ראשונה ל-cache ובעצם הוא ריק. אין מה לעשות לגבי זה, ומכיון שמדוברים הרבה פעוקות זה כמעט תמיד.
2. **Capacity** – אם התוכנית ניגשת למידע בגודל יותר גדול מה-cache. הפתרון הוא להגדיל את גודל ה-cache באשר יודעים מה גודל התוכנית.
3. **Conflict** – מיפוי של שתי מילים לאותו מקום. במקרה זה הפתרון הוא 2-way associativity.
4. **Coherence** – אם יש משותף לכמה תהליכיים אחד מהם שינה את הערך ב-main memory, המילים לא יכולים ערך נכון וצריך לאמת אותן.
5. **Policy** – תלוי בוחרים להכניס ל-cache ואת מי בוחרים לפנות מה-cache. קשה לעשות את המדיוניות האופטימלית.

את מי נשמר ב-cache?

- CPU מודרניים מחולקים ל-2 חלקים: **read allocate & read/write allocate**.
- Cache miss יכול להיגרם מקריאה או מכטיבה של מילה ב-cache.
- קריאה:** אם קראנו מילה לאחרונה נשמר אותה(cache miss). אם קראתי יש סיכוי טוב שאקראי שוב. כלומר קריאה של מילה תגרום לשמירה ב-cache.
- כתביה:** יותר מסובך, יש שתי אופציות –
- כתיבת את השינוי ב-main memory
 - כמו בקריאה, פשוט שמירת המילה ב-cache.

Replacement algorithms

כאשר יש miss יש צורך בפינוי של בלוק מה-cache, הבלוק שמתפנה מכונה הקורבן, **the victim**. הפינוי יהיה במקומות שבו נרצה להכניס בלוק חדש. במיפוי ישיר יש רק מקום אחד זהה, אבל באסוציאטיביות יש בבר יותר אופציות. דרך לבחור את הקורבן היא אקראית, אבל נועד לבחור באופן יותר מכוון שלא יקרה מצב שנפנה בלוק שאחנו משתמשים בו לעיתים קרובות יותר, וכך יש אלגוריתמים המיעודים לכך.

1	4
2	6 cache misses
3	(4 due to cold start)
4	5

Optimal algorithm

הפתרון האופטימלי לבעית החרלה יהיה לפנות בлок שלא נשתמש בו יותר בזמן הקרוב. האלגוריתם שמנמש את זה נקרא אלגוריתם belady. בלתי אפשרי למשמש אותו כי הוא מtabbed על העתיד, אבל הוא ממש להשוויה בינו לבין אלגוריתמים אחרים של cache.

דוגמאות:

נסתכל על סדר הכניסה הבא: 5, 1, 2, 3, 4, 5, 1, 2, 3, 4, 5 כיוון שהוא האחרון ששמש בו. בשנרצה להכניס את 5 ל-cache נזקיף את 4 כי הוא האחרון ששמש בו.

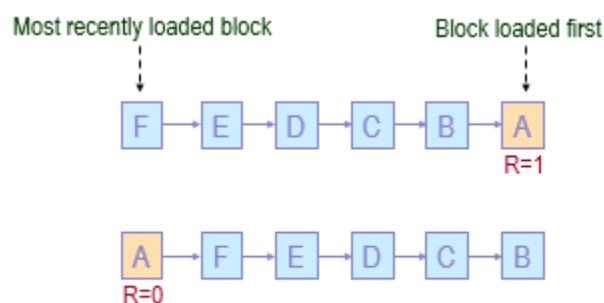
NRU - Not recently used

לפי החקלאות בזמןן, אם השתמשנו בлок עבשו יש סיכוי שנשתמש בו שוב הקרוב. ברגע שנרצה לפנות בлок נבדוק למי מהבלוקים לא השתמשו לאחורה (ביט 0) ואוטו נפנה. לבlok יש רפרנס בית, וכל פעם שיש שימוש בבלוק הבית יהיה 1. פעמי זמן מאפסים את הביטים.

FIFO – First in first out

נשמר את סדר הבלוקים וכשנבחר בקרובן זה יהיה הבלוק שהוכנס ראשון. הבעייה שסדר הכניסה לא בהכרח זהה לתכיפות השימוש בבלוקים. יכול להיות שניגשים הרבה בבלוק cache. כמעט לא נמצא בשימוש. יכול להיות מצב של אונומליה – אם נגדיל את ה-cache misses ירד, אבל אלגוריתם belady הראה שדווקא יכול להיות יותר misses מאשר cache misses באפור, האלגוריתם מתנהג באופן בלתי צפוי:

Cache with 3 blocks	1	2	3	4	1	2	5	5	5	3	4	4
	1	2	3	4	1	2	2	2	2	5	3	3
	1	2	3	4	1	1	1	1	1	2	5	5
time	1	2	3	4	1	2	5	1	2	3	4	4
Cache with 4 blocks	1	2	3	4	4	4	5	1	2	3	4	5
	1	2	3	3	3	3	4	5	1	2	3	4
	1	2	2	2	2	3	4	5	1	2	3	5
	1	1	1	2	3	4	5	1	2	3	4	5



Second chance FIFO

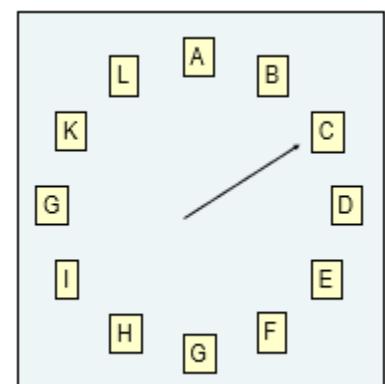
האלגוריתם שייפעל עדין יהיה FIFO, אבל לכל בלוק יש רפרנס בית. כאשר השתמש בבלוק הנפוך את הבית ל-1. הקורבן יבחר מראש התור כאשר אם הרפרנס בית הוא 1, נאפס אותו ונחזיר את הבלוק לסוף התור. האלגוריתם לא ייעיל כל כך כי יכול להויאר מצב שבו נבעור על כל התור בשਬחר קורבן.

אפשרה למימוש היא החזקת הבלוקים כמו שעון, ומחוג השעון יציביע על ראש התור. ברגע שבוחרים קורבן מסתכלים על ההצבעה –

- אם הרפרנס בית הוא 0, ונפנה את הבלוק.

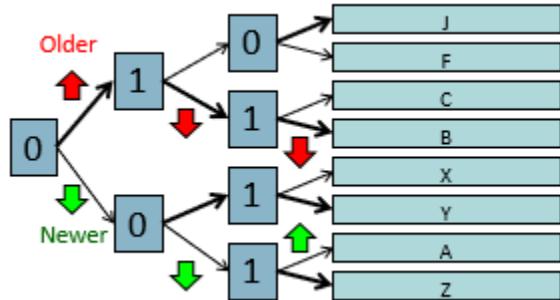
- אם הבית הוא 1, מאפסים את הבית ומעבירים את ההצבעה לבלוק הבא.

אותה פונקציונליות כמו ב-FIFO, רק שבמקום להעביר בлокים העברנו מצביע.



LRU – least recently used

מתבסס על עיקנון הולוקיות בזמןן – אם השתמשנו בבלוק לאחרונה, נראה שהוא השתמש בו שוב. לא נרצה לפנות את הבלוק זהה, והקורבן יהיה הבלוק שהשתמשו בו בזמן הכי רחוק בעבר. קשה למיוש – צריך לשמר את הבלוקים בסדר לפי הזמן שהשתמשו בהם וזה יכול להיות יקר. האלגוריתם יכול להיות בעל-b-caches לא גודלים.



התקורה – $(n_2 \log n)$ ביטים לבлок.

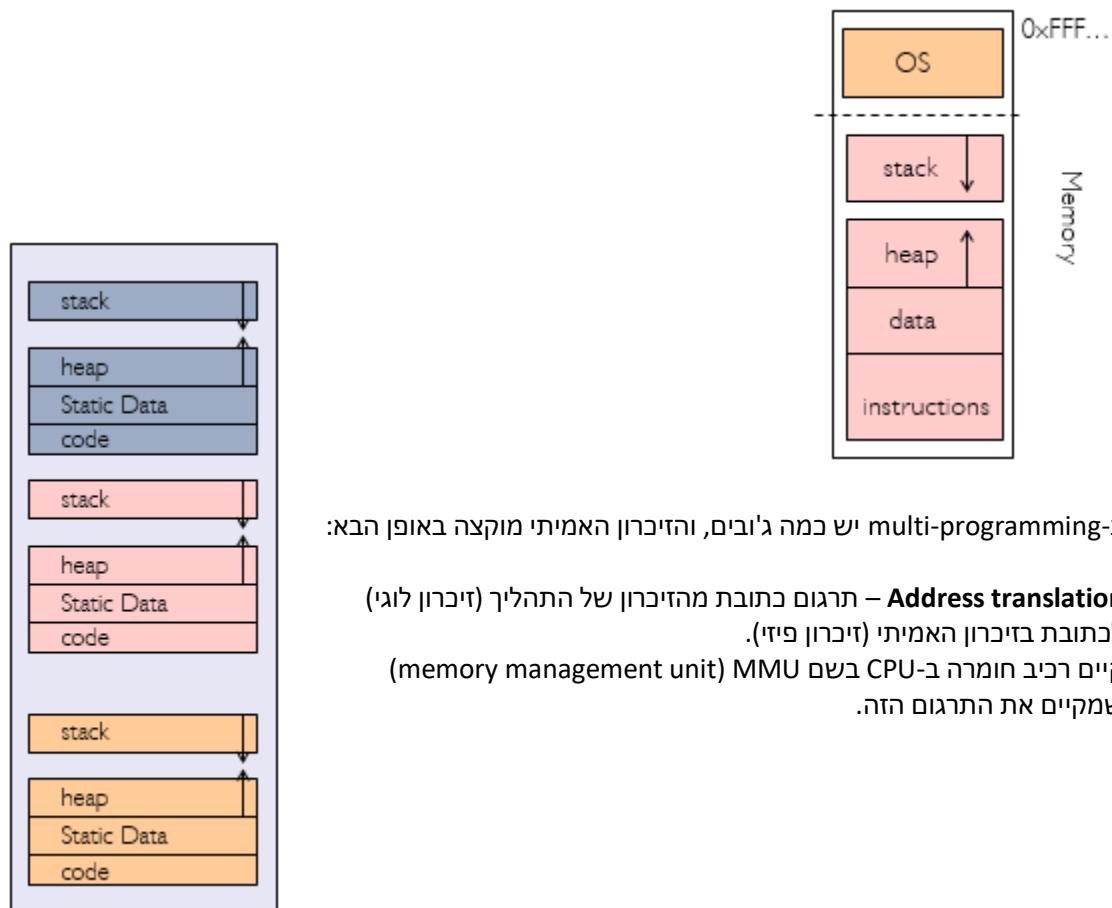
Pseudo-LRU: איזשהו קירוב לאלגוריתם. נחזיק עץ עם ערכאים ביןarios בקידודים שמציעים למי מהבלוקים השתמשו לאחרונה. העלים עצם מצביעים על הבלוקים, וכך לבחר קורבן עוברים משורש העץ לפני הצבעה הישנה יותר עד שמגיעים לעליים (J יותר מאשר M-F, B, יותר מאשר C). דוגמה (במצגת) אם נעקב אחרי ה"אנטי חיצים" נגיע לבlok שהשתמשו בו לאחרונה.

LFU – least frequently used

ספרית כמה פעמים השתמשו בבלוק, והקורבן יהיה הבלוק עם המונה הנמוך ביותר. מצד אחד, אם ניגשנו לבלוק הרבה פעמים סביר שנגש אליו בעוד. מצד שני, אם לא ניגש אליו הרבה עם הזמן יכול להיות שהמונה צריך לדעך.

Memory management

מבחןת כל תהליך הזיכרון בניי באופן הבא:

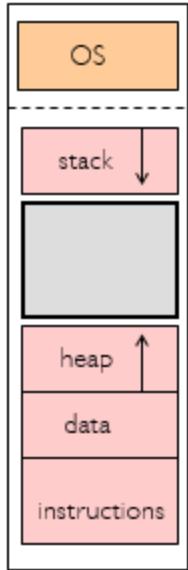


ב-multi-programming יש בינה ג'ובים, והזיכרון האמיתי מוקצה באופן הבא:

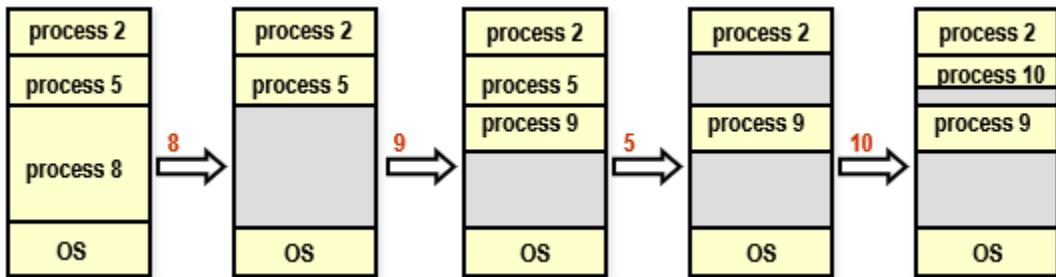
– **Address translation** – תרגום כתובות מהזיכרון של התהיליך (זיכרון לוגי).

לכתובת זיכרון האמיתי (זיכרון פיזי).
קיים רכיב חומרה ב-CPU בשם MMU (memory management unit) שמקיים את התרגומים זהה.

דוגמה לדרך לתרגם שראינו בעבר: כל תהליך מקבל ערך base, והכתובת x בזיכרון הלוגי מתורגם לכתובת $x + \text{base}$ בזיכרון האמיתי.



במה זיכרון מוקצה לתהליך מסויים?
בשלב הקומpileציה ידוע גודל הקוד והדאטה, וגודל heap וה-stack לא ידוע ונרצה להקצות מקום גמיש לבן.
יכולים להיווצר "חורים", זיכרון_UNUSED שלא נשמש בו.
אם נkaza זיכרון לתהליכיים באופן רציף בזיכרון, יכולם להיווצר הרבה חורים שיצטבו לזכרון שלם לתהליך זה יהיה בזבוז מקום – **fragmentation**.



Internal fragmentation – זיכרון פנוי בתוך בלוק שהוקצה לתהליך.

External fragmentation – זיכרון פנוי בין בלוקים של תהליכים שונים. את בעיה זו ננסה לפתור.

פתרונות אפשריים:

- שבירת הזיכרון של תהליך לבלוקים - **pages**.
- **Compaction**

Logical memory

0	a
1	b
2	c
3	d
4	e
5	f
6	g
7	h
8	i
9	j
10	k
11	l
12	m
13	n
14	o
15	p

paging

הזיכרון הלוגי מחולק למיללים, וכל כמה מילים מרכיבותו "עמוד":

הזיכרון הפיזי מחולק **למסגרות** כמו **pages**, וכל עמוד ממופה למסגרת מתאימה לפי **page table** – טבלה המגדירה את אופן המיפוי. לכל תהליך יש טבלה שונה.

מערכת הפעלה אחראית על ניהול זהה ועקבות אחריו המסגרות הריקות בזיכרון כך שאם תהליך רוצה עוד זיכרון – עוד **page**, מערכת הפעלה מקצת לו מסגרת פנوية מהרשימה.

גודל ה-page האופטימלי –

q – גודל ה-page

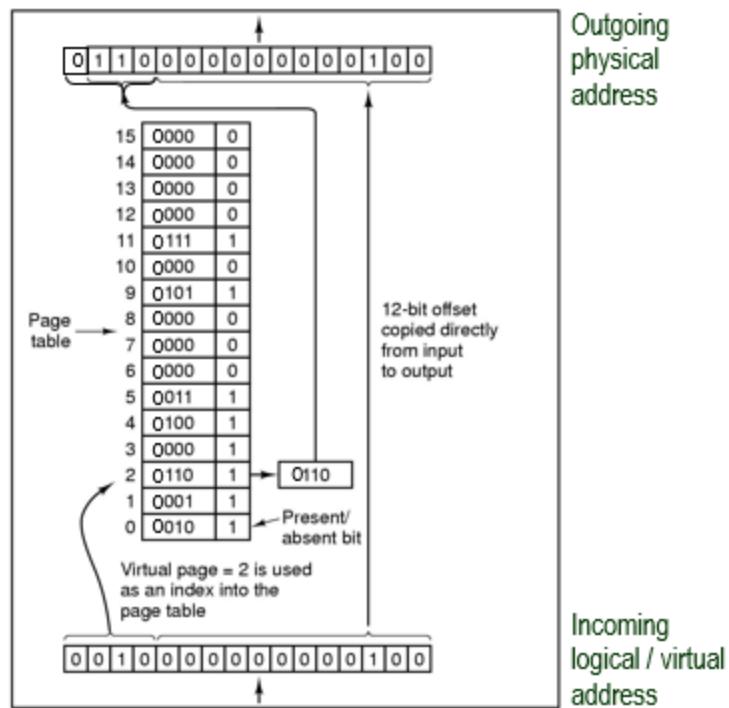
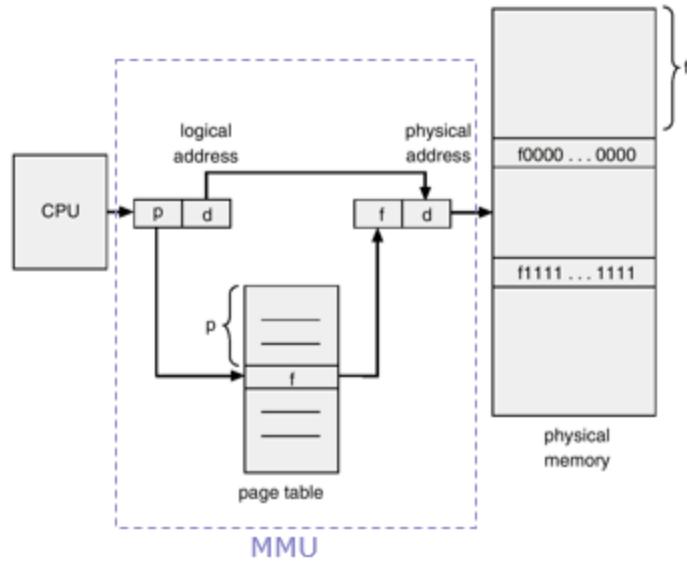
s – גודל התהליך

e – גודל כניסה בטבלת ה-pages

$$\text{התקורה תהיה } \frac{(se/p)}{s/p \text{ entries, each of size } e} + \frac{p/2}{\text{internal fragmentation}}$$

$$\text{הגודל האופטימלי יהיה } \sqrt{2se}$$

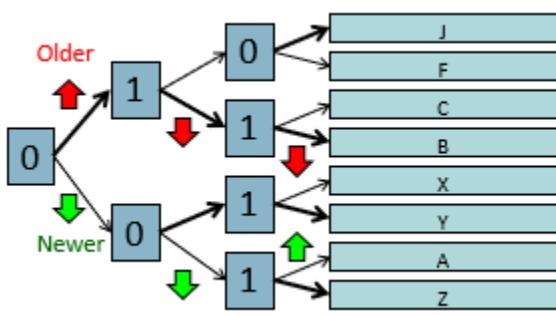
- אם הגודל קטן אז יש פחות פריגמנטציה פנימית, אבל גודל טבלת ה-pages יהיה גדול יותר.



הרצאה 9 – Memory

Addresses Translation

Pseudo-LRU



איזשהו קירוב לאלגוריתם LRU. מעל כל בלוק נוסיף בית של בקרה כבה שבונים עץ שהקדוקדים שלו הם הביטים ומצביעים איפה המקום שאליו נגשנו הći מאוחר.

0 – נגשנו לאחרונה לחצי התחתון והחצית העליון הוא החדש יותר. 1 – להפר. כדי למצוא את ה-LRU נ历 להכין ותיק לפי הצבעה של הקדוקדים מהשורש עד העלים.

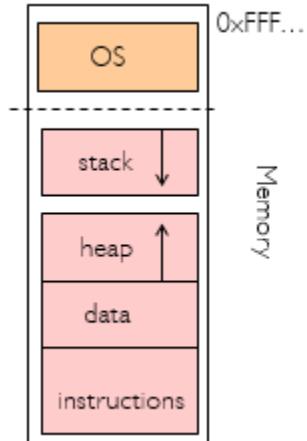
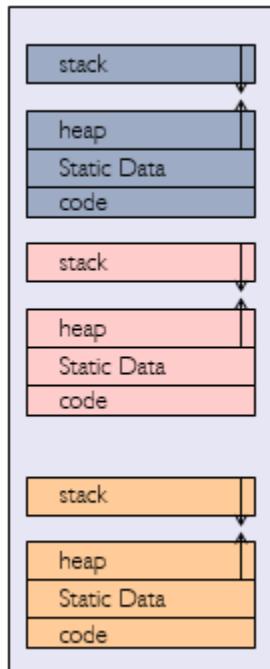
אם בעקבות אחרי ה"אנטי חיצים" – החצים שמראים מי הכי ישן, הגיע לבlok שהשתמשנו בו לאחרונה. (דוגמה ל- Overhead יחסית קטן, בערך בית אחד בלבד, כל שימוש בחומרה.

LFU – least frequently used

ספרת כמה פעמים שהשתמשנו בבלוק, והקורסן יהיה הבלוק עם המונה הנמוך ביותר. מצד אחד, אם נגשנו לבlok הרובה פעמים סביר שניגש אליו בעtid. מצד שני, אם לא ניגש אליו הרבה עם הזמן יכול להיות שהמונה צריך לדעך.

Memory Management

מבחןת כל תהליך כל הזיכרון במערכת שיר ל', הזיכרון בניו באופן הבא:



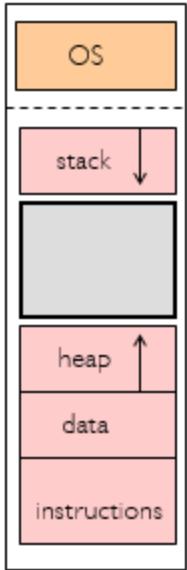
ב-multi-programming יש כמה גיבים, והזיכרון האמיתי מוקצה באופן הבא:

Address translation – תרגום כתובות מהזיכרון של התהליך (זיכרון לוגי)

לכתובת בזיכרון האמיתי (זיכרון פיזי).

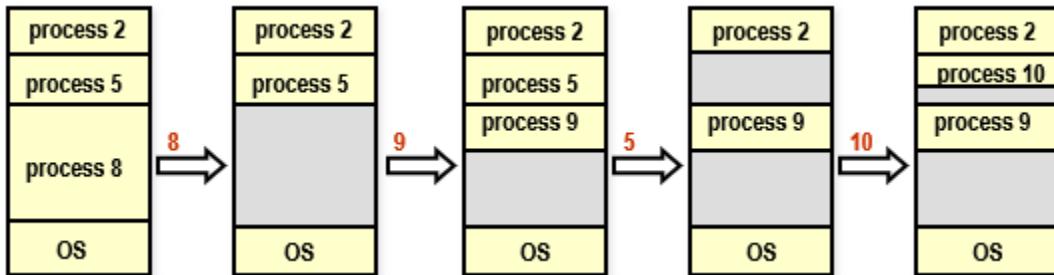
קיים רכיב חומרה ב-CPU בשם **MMU** (memory management unit) שמקיים את התרגומים הללו.

דוגמה לדרך לתרגם שראינו בעבר: כל תהליך מקבל ערך base, והכנתובת x בזיכרון הלוגי מתורגם לכתובת $x + \text{base}$ בזיכרון האמיתי.



במה זיכרון מוקצה לתהליך מסויים?

בשלב הקומpileציה ידוע גודל הקוד והדאטה, וגודל heap וה-stack לא ידוע ונרצה להקצת מקום גמיש לכך שהתהליך יוכל לגודל. יכולם להיווצר מכך "חורים", זיכרון עדף שלא משתמש בו. אם נקצת זיכרון לתהליכיים באופן רציף בזיכרון, יכולם להיווצר הרבה חורים שיצטבו לזכרון שלם לתהליך זה יהיה בזבוז מקום – fragmentation.



– זיכרון פנוי בתוך בלוק שהוקצתה לתהליך.

– זיכרון פנוי בין בלוקים של תהליכיים שונים. את בעיה זו ננסה לפתור.

פתרונות אפשריים:

- Compaction – להזיז בלוקים ולסדר אותם מחדש. כאשר תהליך נגמר מוציאים את שאר התהליכיים בזיכרון. יקר, יש צורך בהעתקת הזיכרון שהוא גם יקרה וגם יתכן שאין מספיק מקום לבן.
- שבירת הזיכרון של תהליך לדפים - pages.

Logical memory

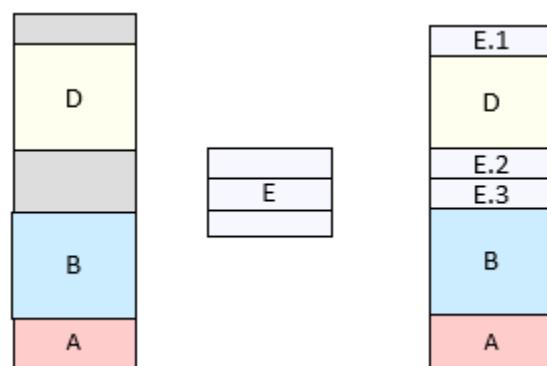
0	a
1	b
2	c
3	d
4	e
5	f
6	g
7	h
8	i
9	j
10	k
11	l
12	m
13	n
14	o
15	p

paging

מסגרת – המיקום בזיכרון הפיזי בו שמינית הדף

הזיכרון הפיזי מחולק למסגרות, וכל עמוד ממוקפה למסגרת מתאימה לפי page table – טבלה המגדירה את אופן המיפוי. לכל תהליך יש טבלה שונה.

מערכת הפעלה אחראית על הניהול הזה ועוקבת אחרי המסגרות הריקות בזיכרון כך שאם תהליך רוצה עוד זיכרון – עוד page, מערכת הפעלה מקיצה לו מסגרת פנوية מהרשימה.



גודל-page האופטימלי –

- ק – גודל-page
- ס – גודל התהליך
- ה – גודל כניסה בטבלת pages

$$\frac{(se/p)}{s/p \text{ entries, each of size } e} + \frac{p/2}{\text{internal fragmentation}}$$

התקורה תהיה

$\sqrt{(2se)}$

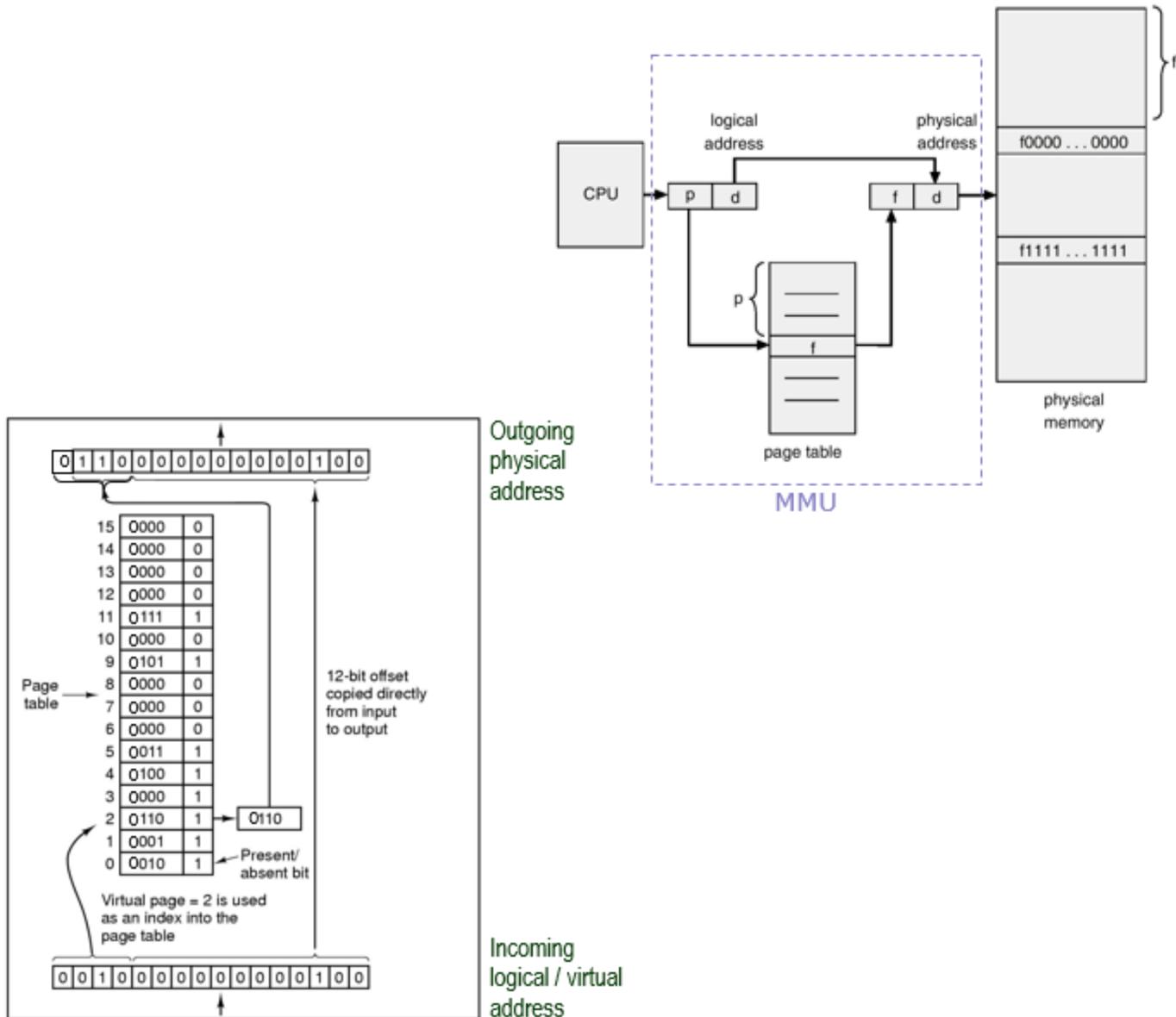
- אם גודל הדף קטן אז יש פחות פרוגמנטציה פנימית, אבל גודל טבלת pages יהיה גדול יותר.

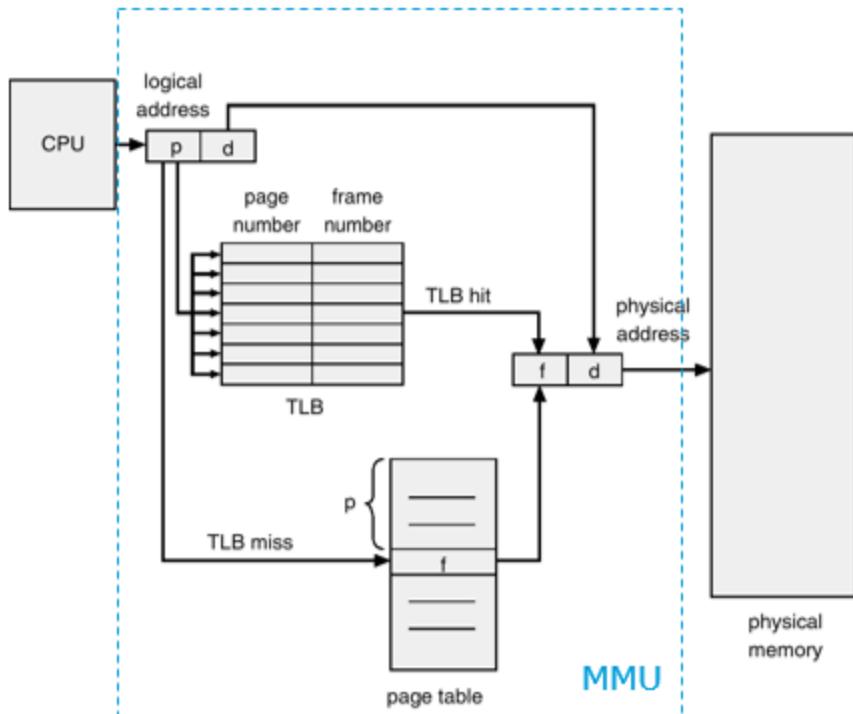
מיפוי בתיבות

במיifié, הכתובת בזיכרון הלוגי יכולה להיות מחלוקת לשני חלקים:

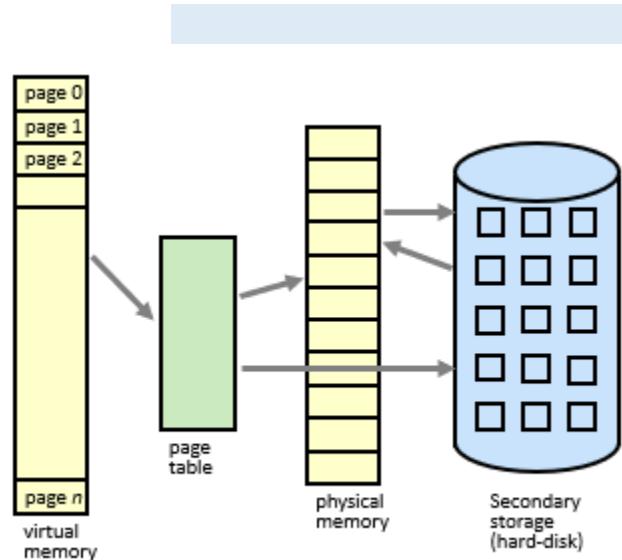
- Page - איפא בטבלה כתובה המסגרת בזיכרון הפיזי.
- Offset - באיזה מקום במסגרת הכתובת נמצא.

לדוגמה המיקום של הכתובת 00110 יהי בשורה 001 בטבלה, ובאידקס 10 במסגרת הכתובת באותה שורה.





TLB
עם המיפוי הגישה ל זיכרון מוכפלת, ניגשים פעומים והתקורה עצומה. אך יש מיחד לטבלת הדפים - **TLB** - Translation Lookaside Buffer המיפוי של הדף לمسגרת נשמר ב-cache.



Virtual memory

בל תהליך חושב שיש לו את כל הזיכרון. בערךון מספר התהיליכים לא מוגבל וכן באופן כללי הזיכרון הלוגי יותר גדול מהפיזי.
זיכרון וירטואלי – רק חלק מהמידע של התהליך יהיה בזיכרון הפיזי, כי לא משתמשים בכל הזמן. השאר יהיה ב-cache או רמה אחת מתחת להיררכיה. הזיכרון הראשי הוא סוג של cache עבור הディיסק.
עבור כל דף יהיה בטבלה בית שיצין אם הדף נמצא בזיכרון הפיזי.

מה שומרם ב-cache? (main memory) cache
בהתאם לדילמות ה-cache שדיברנו עליה, מימוש גישה מהירה יעשה ע"י TLB והטבלה. מה נהרצה לשומר ב-cache במקרה של דפים? שתיאר אופציונות:
1. **Page demanding** – כאשר דף מסויים לא נמצא בזיכרון הפיזי נעלם אותו. "תכנו הרבה Page faults".
2. **Pre-paging** – מערכת הפעלה מנחשת מה יהיה הדפים שנשתחמש בהם. אם מערכת הפעלה מנחשת נכון ונחסך זמן.

Page fault

בשדה לא נמצא בזמן שההlixir מנסה לגשת אליו יש trap בשם **Page fault**, התהליך נכנס למצב המתנה עד שהדף יעלה. מערכת הפעלה מורה ל-DMA שבדיסק להעתיק את הדף לזכרון המרכזי ועד שהוא מסיים התהליך במתנה. המחיר:

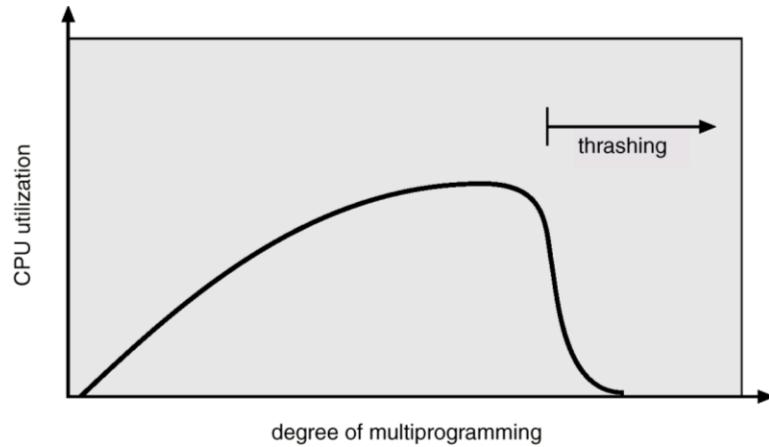
Access time – כמה זמן לוקח לגשת לזכרון בממוצע:

$$p(\text{page fault time}) + (1 - p)(\text{memory access time})$$

P – ההסתברות ל-page fault

Slowdown – בכמה זמן המערכת הואטה בתוצאה מכך שלא הצל שומר בזיכרון המרכז:
Effective access time / memory access time

CPU usage – הזמן שבו ה-CPU מבצע הוראות. באשר יותר תהליכים עובדים במקביל כך ה-**usage CPU** עולה. מאיזושהי נקודה הוא צונח וזאת מכיוון שבנקודה הזאת יש יותר מדי תהליכים וכך הזיכרון שדרוש לעיבוד שלהם גדול מהזיכרון המרכזי. יש יותר מדי **page faults**. ה-CPU עסוק בהחליפ דפים. תופעה זו נקראת **thrashing**.



הפתרון – הגבלה של multi-programming ע"י השעיה של תהליכים אחרי במות מסוימת.

הרצאה 10 – Memory

Page Table

טבלת דפים היררכית

דיברנו על כך שטבלה הדפים יכולה להיות גדולה יותר מהזיכרון. טבלת דפים היררכית מהוות פתרון לבעה זו כאשר הטבלה מחולקת לתתי טבלאות קטנות יותר. ישנן שתי רמות של הטבלה:

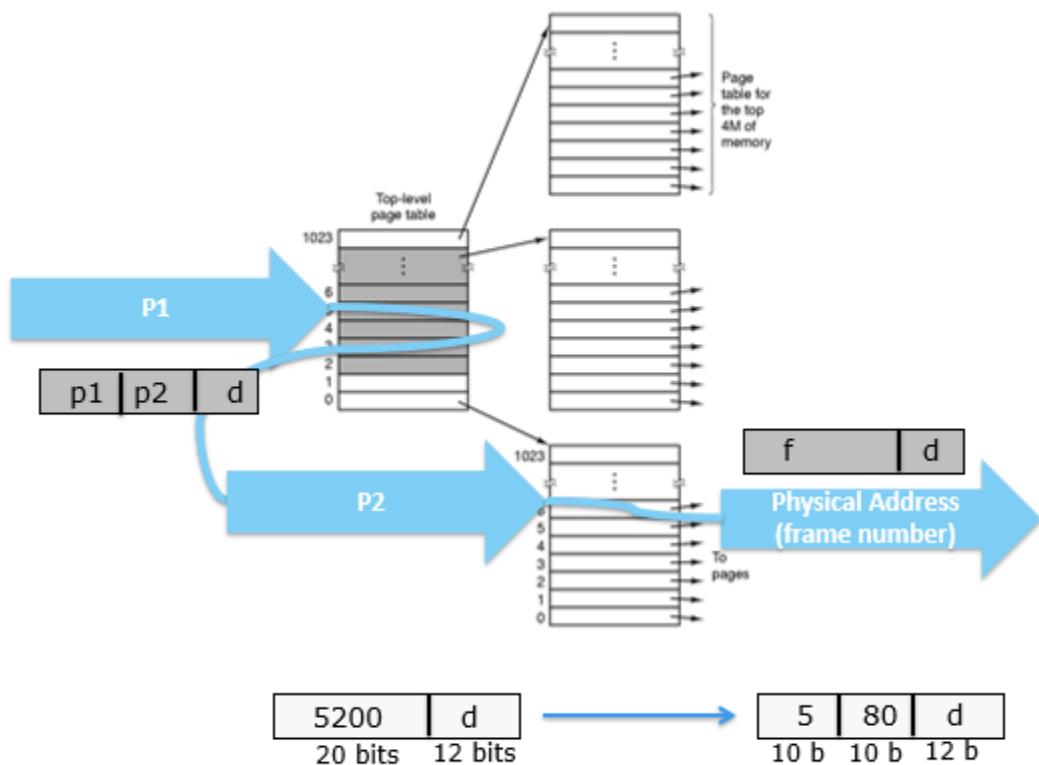
.1. **Top level page table** אליה הדיכון הלוגי ממופה ישירות.

.2. **Second level page table** טבלה המחולקת לשליישות. כל עמוד ב-level top ממופה לשליישיה ולפיה יודעים באיזו מסגרת לבדוק בזיכרון הפיזי.

ככה, במקרה להציג טבלה אחת עם מיליון שורות ו-1000 דפים (במקרה שבו יש 32 ביט). במקומות זה נחזק 1001 טבלאות (ה-1 זה ה-level top) עם 1000 שורות כל אחת.

אם מיקודם חילקו את בתובת הדף לשני חלקים כאשר חלק אחד הוא המיקום בטבלה והחלק השני הוא ה-offset, offset, העשוי לsecond level.

offset נשאר אותו דבר ונקבע לפי גודל העמוד.

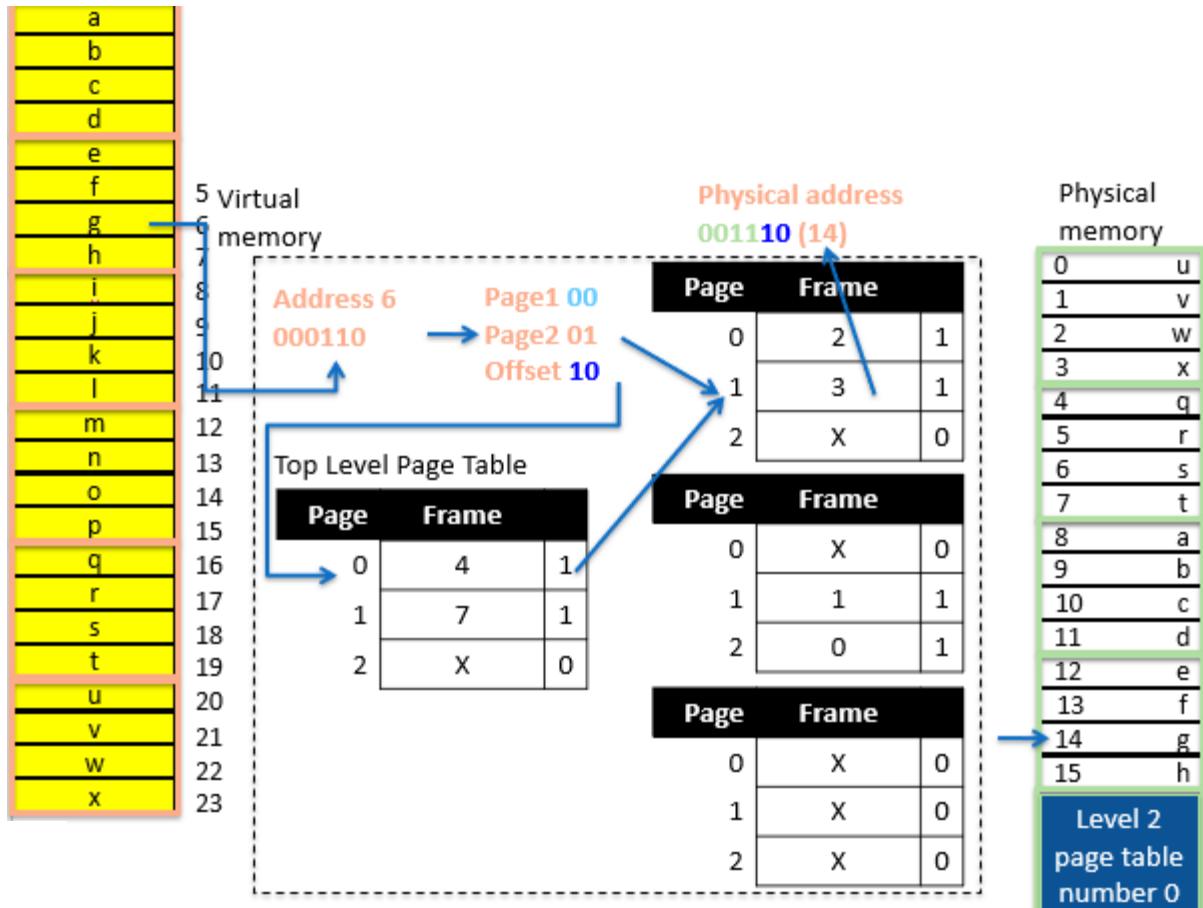


דוגמא:

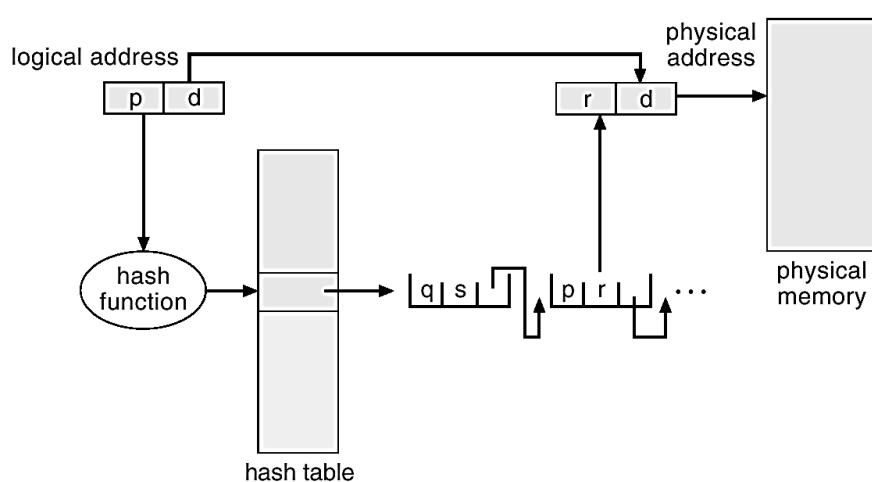
הכתובת 6 בטבלה ממופה באופן הבא:

00 01 10

נבדוק בטבלה ה-level top במקום **00** באיזו טבלה לחפש ב-level second. במקרה זה צריך לחפש בטבלה הרביעית ושם נסתכל על המיקום **01** لأن הכתובת ממופה בזיכרון הפיזי. בדוגמה כתוב שהמיפוי הוא למסגרת 3 בזיכרון הפיזי ונחפש שם לפי האפסט במיקום **10**.

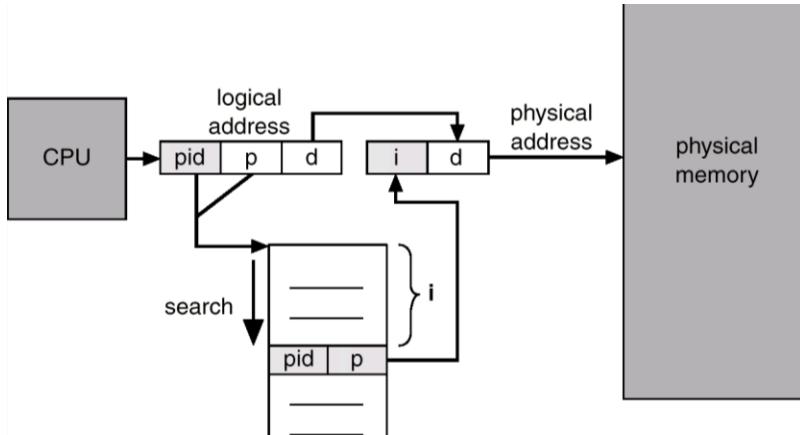


בטבלה היררכית – כאשר משתמשים בטבלה היררכית ישן שלוש גישות ליזכרון. בגישה top לא יהיה אף פעם fault page, אבל בשתי הגישות האחרות בן יכול להיות ואולי אחד אחריו השני. רוב ה-second level page tables הן ערבי זבל, ריקות, והן צריך לאחסן אותן. כאשר ניגש אליה, במקום שיהיה page fault נקצת דף של אפסים בשביב הטבלה?



Hash table
באשר מחלקים את הכתובת ל-page ו-offset,
ה-page מתרגם לאיזשאו אינדקס שיצין את
המיקום בזיכרון הפיזי. ה-page מתרגם ע"י
פונקציית גיבוב למיקום בטבלה הגיבוב בו יש
רשימה מוקשתת שמקילה את המיפוי ליזכרון.
הפיזי של כל דף שמומפה לאותו מקום בטבלה.
בדוגמה ק מתרגם ל-r.

זמן ריצה בתוכנת הוא O(1) אבל יתכונו
בטלאות או רשימות מאד ארוכות שייעלו אותו,
בעיקר בשיש page fault שבמקרה זה נדרש
לעבור על כל הרשימה.
לא פופולרית כל כך 😞

Inverted page table

הופכים את הסתכולות על המיפוי – נשמר לכל פריטים בזיכרון הפיזי איזה תוכן נמצא בו, איזה עמוד של איזה תהילך. בשנורצה לבדוק page מסוים, נסתכל בכל המסגרות זיכרון הפיזי עד שנגיע למקום הנכון.

- זמן ריצה לינארי.
- השיטה יעללה בשזה זיכרון הפיזי מאוד קטן כי המעבר מהיר.

הבל ביחס

ראינו שבבינן בתובות וירטואלית וטבלה היררכית בגודל 2, נתרגם את הכתובת לתובות בזיכרון הפיזי ע"י פירוקה ל-**P1, offset**.

TLB – נזכר שה-TLB הוא cache של טבלת הדפים שם יש מיפוי מה-page ל-frame. החיפוש ב-TLB לא מתיחס להיררכיה וקורא במקביל. החיפוש בודק האם ניתן לתרגם את אותו page למסגרת, ואם כן מدلגים על החיפוש בטבלאות היררכיה ויש נגשים לזכרון הפיזי. ברוב המקרים זה אכן מה שקרה. אם יש miss TLB ניגש לזכרון הראשי.

ב-cache – ב-page – Cache שמרוות בתובות פיזיות. אם הכתובת הלוגית שמורה ב-cache נתרגם אותה (את החלק של ה-page offset נשאר כמו שהוא), את התרגום נחלק כמו שראינו בשינויים ל-cache, cache miss. אם יש hit לא ניגש בכלל לזכרון הראשי ואם miss בן נחפש שם.

Parallel TLB – כדי לזרז את כל התהליכים נרצה שה-TLB וה-cache יעבדו במקביל כאשר יש hit .tlb. אם התג היה בגודל של-page זה היה אפשרי: תרגום page-to-frame במקביל לשילוף הבלוק בהתאם לתג. אם התג יתאים לחלק במסגרת עליון נסתכל דרך תרגום הכתובת הלוגית, יש hit .cache miss. אחרת .cache miss. את הכתובת המתורגמת נחלק כמו שעשינו עד בה עם cache. אם כל האינדקס נמצא בחלק של האופסוט של הכתובת הלוגית בין page-to-frame לא ישנה את האינדקס. נחפש בין התגים האפשריים איפה יכול להיות hit .cache miss. במקביל נתרגם את page-to-frame ואם התג שמצאנו זהה למסגרת שקיבלו יש hit .cache hit. אם לא, נמשיך לחפש בתגים האחרים.

לסיבום, הסתכלנו על שלוש בעיות הקשורות לניהול זיכרון:

Problem	Solution
Process view is different than actual view	Address Translation: base+bound, page tables
External fragmentation	Compaction, paging
Logical memory is larger than physical memory	Virtual memory + swapping

אלגוריתמי החלפה בזיכרון הלוגי**בעיות בניהול זיכרון**

דיברנו על כך שהזיכרון הלוגי הוא כמו cache עבור הディיסק, וכך עם אותן בעיות שהתמודדו בניהול ה-cache נצטרך להתחמוד גם עבשו:

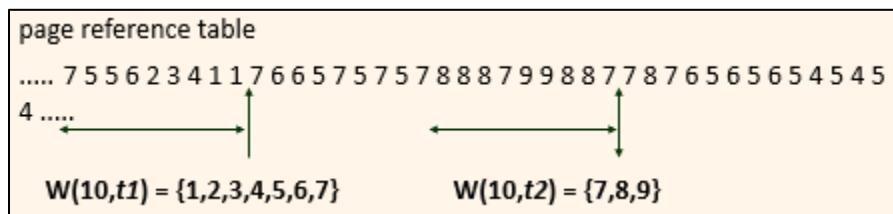
1. **תרגום כתובות – TLB & page table.**
2. **מה נשמר? לפि demand paging.**
3. **מה נמחק? אלגוריתמי החלפה שלמדו תקפים גם כאן, אבל ישנים אלגוריתמים נוספים שמתאימים יותר לזכרון הלוגי.**

Working set

תהליך מסוים עובד ב- k גישות לזכרון על סט מסוים של pages ולא מעבר – working set. הлокליות בזמן של היא כמה פעולות, כמו עמודים התחילה התייחס ב- k פעולות האחרונות בזמן מסוים?

בשנecer קרובן נבחר דף שלא ב- k :

- אם k קטן מדי יוכל להיות שנעיף pages שנבחר בזמן הקרוב והוא page fault.
- אם k גדול מדי בnearest pages שלא השתמש בהם.



האלגוריתם צריך לחשב את ה- k הטעינה הקבועות. מ עבר לזכרון הטעינה צריך לזכור מתי הגיע לכל דף בפעם האחרונה:

$$w(k, t) = w(k, t - 1) \setminus \{ \text{reference } t - k \} \cup \{ \text{reference } t \}$$

באופן כללי יהיה קל יותר לחשב את הסט כתלות בזמן (הדף שביביגנו אליו מילוי שניות האחרונות)

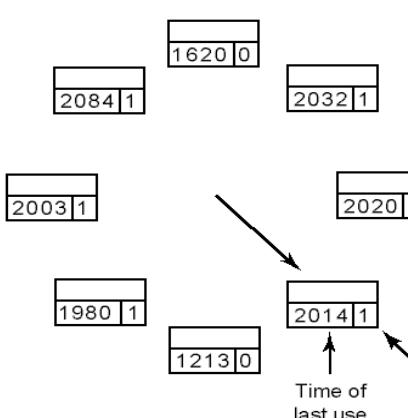
WSClock

נסדר את כל הדפים בתהlixir באיזה דף נמצא, בנוסף, בנוספ' ל:

- מצביע שמצין באיזה דף נמצא.
- שעון וירטואלי.

לכל דף יהיה ערך שיצין את הזמן האחרון שבו השתמשו.

כל פעם שניגש לעמוד נעלם את הרפרנס ביט מ-0 ל-1.



פעם בכמה זמן תהיה פסיקת שעון, בה:

נסתכל על הזמן הוירטואלי ונעדכן לכל עמוד שבו הביט הוא 1 וונעדכן את

הערך שלו לזמן הוירטואלי שצווין בעת הפסיקה.

נעלה את השעון הוירטואלי.

-
-
-

כאשר יש page fault וצריך לפנות את אחד הדפים:

נסתכל על הערך בדף אליו החץ מצביע.

אם הרפרנס ביט הוא 1 זה אומר שניגשנו אליו אחרי פסיקת השעון

האחרונה, כלומר השתמשנו בו לאחרונה, ונקדם את המצביע.

- אם הרפרנס ביט הוא 0 נבדוק אם הזמן בשעון הווירטואלי פחותו מ-k. אם כן, השתמשנו בו ב- k שניות האחרונות ולא נרצה לפנות אותו.

(current virtual time) – (time of last use) $< k$

- אם לא, נפנה את הדף.

דףים מולכלים ודףים נקיים

- **דף מולכלר** – דף שונה מאשר מפעם אחרת שניגשנו אליו מהדיסק. בשנרצה לפנות אותו קודם נctruck להעתיק את הכתובת שלו על הדיסק ורק אז להעלות דף חדש ל'יבرون'.
 - **דף נקי** – לא כתבנו בו, ניתן לבתו מישחו עליי.
- בשנרצה לפנות דפים האלגוריתם WSClock יעדיף לפנות pages נקיים כאשר לא ניתן אף אחד לפנות. אם האלגוריתם יתקל בדף מולכלר ושין, הוא יודיע למכרך הפעלה להתחיל להעתיק את הדף לדיסק בר שיחוף לנקי ונוכל לפנות אותו בפעם הבאה.

Global vs. Local paging

- **Local paging** – כאשר תחיל רוצה להעיף pages ולהביבס במקום הוא עיף pages שלו.
- **Global paging** – התחילה עיף pages של תחיל אחר.

הרצאה 11 – Files

introduction

דיברנו על בר שמערכת הפעלה מקשרת בין החומרה והאחסון לבין המستخدمים והאפליקציות, כמו כן חזכרנו שהdisk לא נדף – גם כשהוא כבוי המידע בו נשמר, וכן הוא משמש כגבוי לזכרון אבל בשמנטים לגשת למידע שבו זה לוקח יותר זמן. לעיתים אפליקציות יעדיפו לשמור בdisk מידע ולא-*main* מידע main מהסיבות הבאות:

- בשמהדר בדאתה של יותר מגודל הזיכרון ואין לכך מספיק מקום (גודל הזיכרון מוגבל).
- באשר מביבים את המחשב הדאטה שבדיכרונו נמחק.
- תהליכיים שונים ירצו גישה לאותו מידע.

לכן נעדיף לשימוש במערכת אחסון שתמלא אחר שלושת הקriterיוונים הבאים:

1. יכולת לאחסן ב莫ות גדולה של מידע.
2. המידע ישאר גם אם התהיליך שימושתו בו הסתיים.
3. הרבה תהליכיים שונים יכולים לגשת למידע אחד.

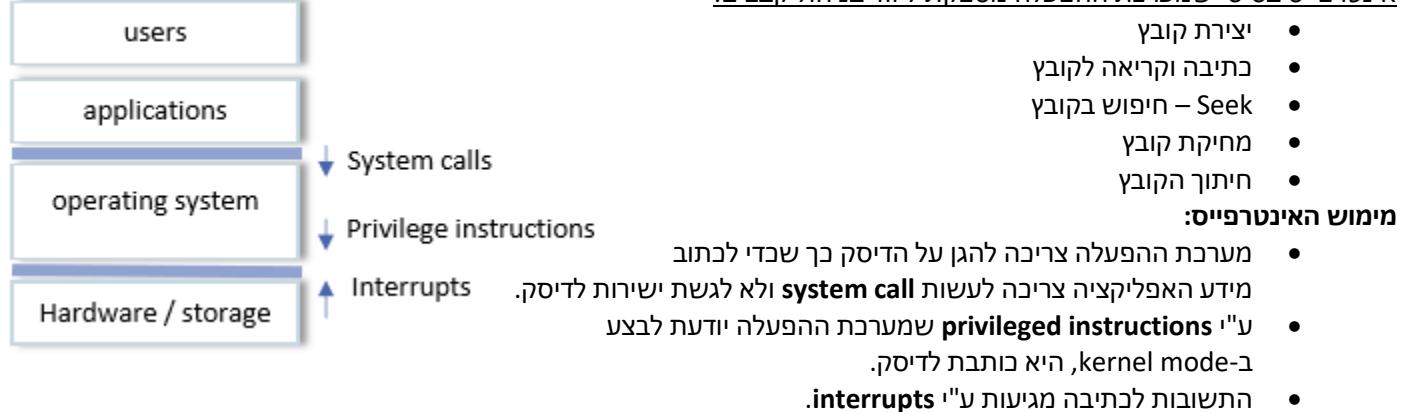
הפתרון לכך הוא לאחסן את המידע בdisk ולארגן אותו בצורה אבסטרקטית בקבצים – files, שימושתפים בין תהליכיים ומונחים ע"י מערכת הפעלה ונשמרת על מידע ששנשמרת על מערכות אחסון יוכל להכיל כל סוג של מידע

קובץ – יחידה לוגית של מידע שנשמרת על מערכות אחסון ויכול להכיל כל סוג של מידע

בשנדבר על מערכת קבצים נסתכל על שני איס派קטים שונים:

1. מה הוא האינטראפיס שמערכת הפעלה נותנת לאפליקציות והיוזרים כדי לנהל את הקבצים? איזה system calls אפשר לעשות כדי לנהל את הקבצים? למשל `open/close`, `read/write`, `directories`.
2. איך מערכת הפעלה ממשתתת את ניהול הקבצים?

איןטראפיס בסיסי שמערכת הפעלה מספקת לסייע בניהול קבצים:



File attributes

שם:

- כאשר יוצר קובץ הוא נותן לו שם. דרך השם יזרים או תהליכי אחרים יכולים לגשת למידע זהה.
- הקונבנציות לשם תלויות במערכות הפעלה.
- ניהול השמות וניהול הדאטה לא זהים.

Identifier – מהה ייחודי לכל קובץ

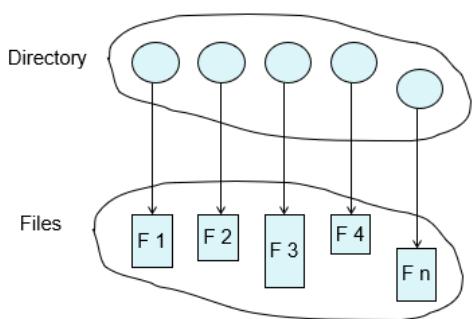
סוג

מקום

גודל

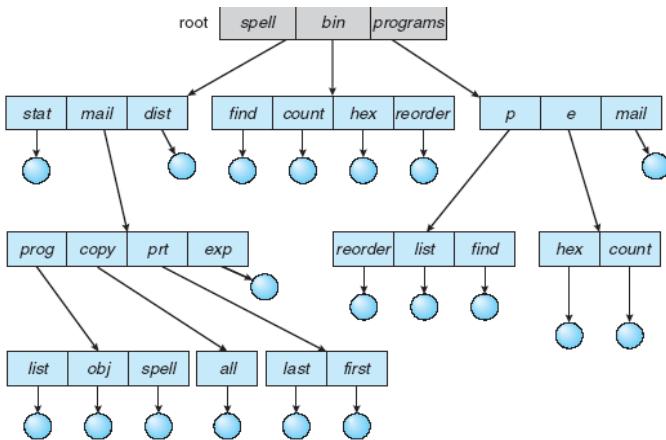
הרשאות: גישה לבתיחה, גישה לבתיחה, גישה לביצוע.

משתנה בין מערכת הפעלת אחת לאחרת.
– דיהוי הסוג שבו משתמשים. **File extension**

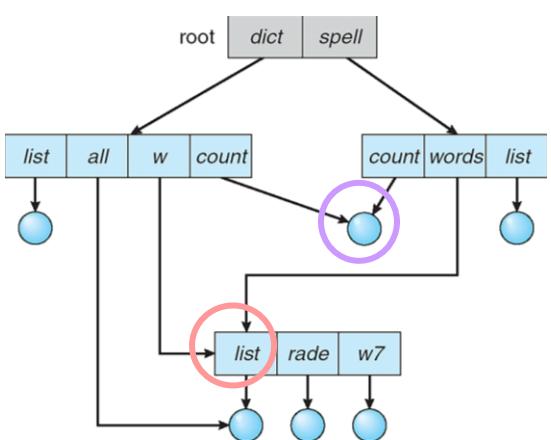


הקבצים מאורגנים בספריות שבהן ייעילות המשתמש. **Directories** – כל directory/ספרייה ממומשת ע"י איזשהו קובץ שמכיל רשימה של קבצים וניתן לבצע דרכה את הפעולות הבאות מובילו להיכנס לקובץ עצמו:

- להסכלל על כל הקבצים בספריה
- למצוא/למחוק קבצים
- לשחרר
- לשנות שם
- וכו'



בד"כ הספריה בנויה בצורה עצ בגובה שרירותי, כל רשותה כוללת או ספריה בפני עצמה או דאטה (קובץ). המטריה של העץ היא להבין איפה נמצא הקובץ. בסוף העץ יש מצביע לדאטה וכן לשנות דברים הקשורים לשם או למקומות מספיק לשנות את העץ.



Hard links vs. symbolic links
לפעמים הגրפים האלה הם ממש עצים אלא א-ציקליים, ניתן להציבו במקום מקומות שונים לאוטו תא.
זה דוגמה, אם נרצה לקשר את dict/spell/count ואת dict/count/spell ב� ששנייהם יצביעו לאוטו דאטה, השתמש בפקודה הבאה:

In spale/count dict/count

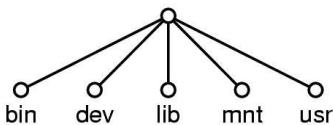
כאשר נציג את אחת ההצביעות על הדאטה (למשל נחליף את dict/count ב- dict/count1) הוא לא ישתנה (דוגמה בשקופית 18).

Symbolic link – קיימים שני מצביעים לאוטו שם. הקישור של שני מצביעים שונים נעשה באמצעות פקודה. במקרה זה, כאשר נציג את השם, נחליף את השם, ההצביעת אל הדאטה שאליו מצביע list תמחק וכך dict/w/list/w7.

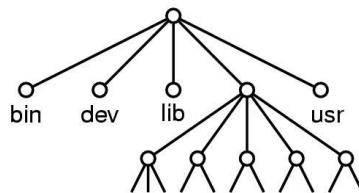
פתרונות:

1. שומרים את כמה המצביעים לקובץ ב-reference count, ואם המצביע לקובץ נמחק המונה יוד. אם הוא הגיע ל-0 מוחקם את הקובץ.

.2 – הוספת מערכת קבצים אחת לאחלה.



(a)



(b)

File protection

הגנה על קובץ מיוזרים אחרים לפי החלטת היוזר – מי ניגש ומה הוא יכול לעשות. ניתן להסכל על הרשותות עבור יוזר יחיד או עבור קבוצות של יוזרים.

סוגי הרשותות:

Read	•
Write	•
Execute	•
Append	•
Delete	•
List	•

שתי גישות למתן הרשותות:

1. בלינוקס יש שלושה סוגי יוזרים שונים לתת להם הרשותות:

- Owner – היוזר שיצר את הקובץ.
- Group – קבוצה שבה היוזר נמצא.
- Public – כלום.

ונitin לתת את הרשותות הבאות: read, right, execute. בビיאריה יש 8 קומבינציות של הרשותות.
לדוגמה, ניתן את הרשותות:

		RWX
a) owner access	7	\Rightarrow 1 1 1 RWX
b) group access	6	\Rightarrow 1 1 0 RWX
c) public access	1	\Rightarrow 0 0 1

ונריץ את הפקודה הבאה:

chmod 761 game

הבעיה: רק שלוש סוגי הרשותות ושלושה יוזרים.

2. בוינדוס משתמשים בגישה **access control list**ACL שלכל יוזר ניתנות הרשותות בנפרד, ניתן להגדית בה להחולצות יותר נמנוכות אבל יותר מסובך למש.

גישה לקבצים**איך קוראים מידע?**

שתי דרכים שונות לקריאת מידע:

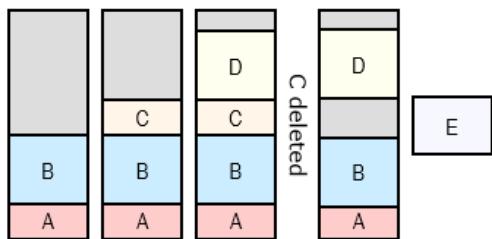
1. **גישה רציפה / sequential access** – מההתחלת עד הסוף. כאשר נרצה לחדור לשמהו שקרנו או לкопץ להתחלה זה יהיה יותר יקר.
2. **גישה רנדומלית / random access** – המידע מפוזר, DRAM למשל. מערכות קבצים בד"כ מומושות בגישה רציפה.

דילמות במימוש מערכות קבצים

1. מהו מבנה הנתונים בו כדאי לשמור את הדטהה בדיסק?

2. איך לשמר?

3. תמייה בגישה רציפה אבל גם ברandomilit בקרה הצור.

**프로그램נטציה**

גם כאן, מתקיים שברור באשר שomersים קבצים בגודל אחר ונותר חור בזיכרון שלא מספיק כדי להכיל את הקובץ הבא שנרצה לאחסן.

פתרון 1: compacting, יקר ולא יעיל.

פתרון 2: paging. שבירת קובץ רציף לבLOCKים.

בثور סקטורים וגם בגין-blocks-system view – אוסף של בלוקים. המידע שמור בבלוקים שנשמרים על הדיסק כדי למצוא byte בודד צריך לגשת לבлок בו הוא נמצא.

כדי לגשת לקבצים נשמרים על הhard disk צריך לשיטים על ה-

tract הנכון את הזרוע המגנטית, וצריך לחכות שנגיע לסקטור הנכון, ואז ניתן לבצע את הפעולה.

- זמן המעבר מ-track ל-track נקרא seek time וлокט.

כ-12-8 מילישניות.

- סיבוב הדיסק נמדד ע"י סיבוב בשנייה RPM.

כאשר נארגן מידע על הדיסק נרצה לארגן אותו בסקטורים קרובים או לפחות באותו tract כדי לצמצם את הזמן הזה –

לכן כדאי לשמר את הבלוקים של הקובץ בזורה רציפה – contiguous allocation.

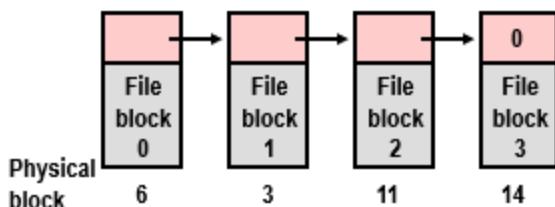
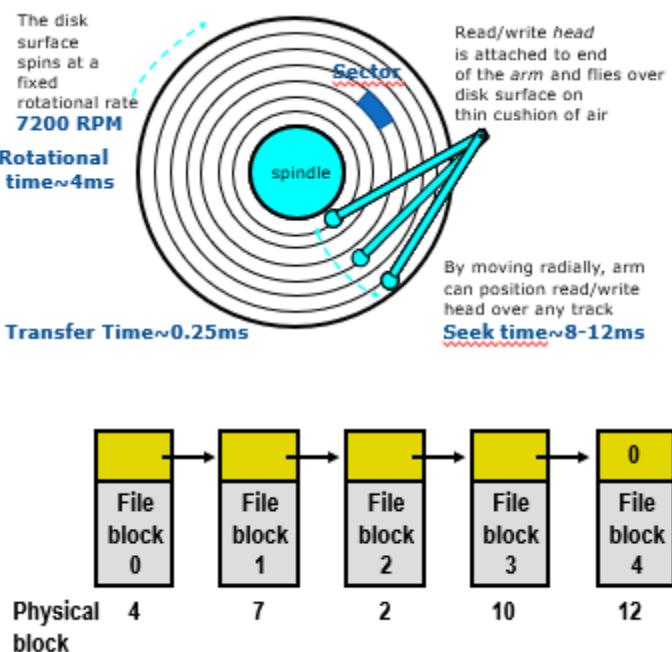
אבל יתרך מצב של פרגמנטציה.

פתרון: שמירת הבלוקים בראשיה מקושרת בר' שככל בלוק מצביע על הבלוק הבא.

בעיות:

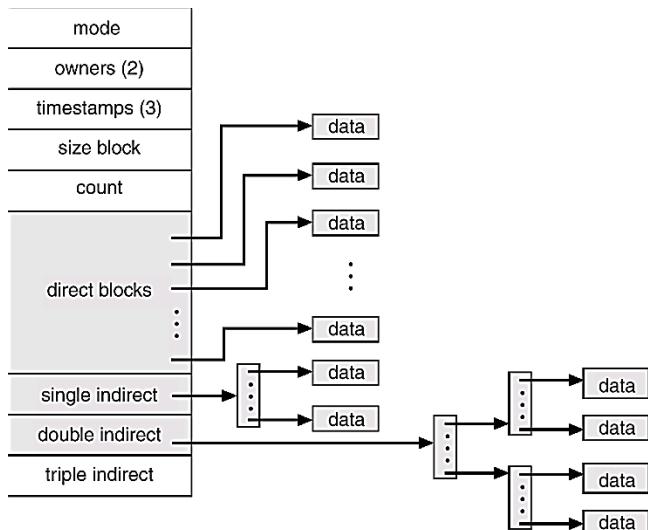
1. גודל הבלוקים הוא ברור לא בחזקת 2 כי הפונטטים מוסיפים להם תוכן.
2. הגישה הנעשית באן היא גישה אקרואית ויכולה להיות איטית.

הפתרון: טבלה שבה המידע נשמר בזיכרון ומנהל את הקבצים.



0
1
2 10
3 11
4 7
5
6 3
7 2
8
9
10 12
11 14
12 -1
13
14 -1
15

File Allocation Table
(Stored in Memory)



FAT – File Allocation Table

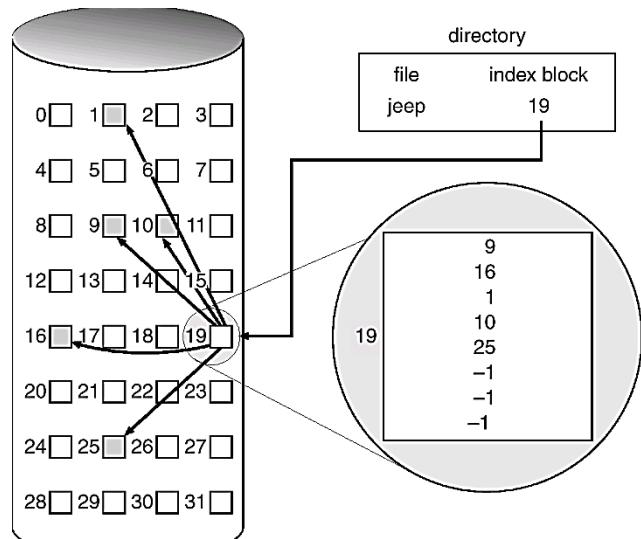
בטבלה מצין איפה נמצא כל קובץ. באינדקס של כל בלוק מצין האינדקס הבא אליו הבלוק מציביע.

היתרון:

1. גודל הבלוקים קטן והוא בחזקת 2.
2. הטבלה נשמרת בזיכרון הראשי ולא בדיסק, לכן הגישה האקרואית תהיה מהירה יותר.

הבעיה: לעיתים הטבלה תהיה עצומה ולא יהיה לה מקום בזיכרון.

הפתרון: לא נאחסן את כל הטבלה בזיכרון – **index allocation**.
לכל קובץ יש בלוק בתחילתו שבו מאוחסנים כל האינדקסים בטבלה שקשורים אליו.
תהייה חלוקה של שלוש רמות של הצבעות.



הרצאה 12 – Files Scheduling – 12

File systems

פתרונות קבצים

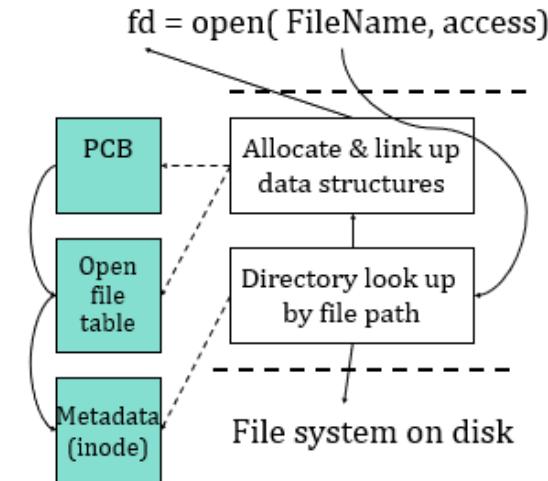
לכל תהליך יש קישור לקבצים אליו הוא יכול לגשת ומערכת הפעלה מאפשרת את האבstrטקציה הזאת וניגשת לקובץ ע"י File descriptor שהוא מצביע למיקום המידע שיש בקובץ. PCB יש טבלה ובה מידע על הקובץ (הרשאות, שם וכו'), האינדקס של הקובץ בטבלה הוא בעצם ה-fd. הפעלה היא שכאשר מספר הפעולות שונים פותחים אותו קובץ יכול להיות חסר סנכרון ושבפול מידע. הפתרון לכך בلينוקס הוא שمرة של שלוש טבלאות שכל אחת מצביעה לשנייה:

1. **Per-process table (within the PCB)** – רשימת הקבצים שה-PCB פתח. לכל קובץ יש מצביע (ולא מידע) לטבלה מסווג system wide table.

2. **System wide table** – מחזיקה את כל הקבצים הפתוחים. כל פעם שקובץ נפתח נפתחת לו שורה בטבלה. בנוסף הטבלה מחזיקה את ה-offset של אותו תהליך שפתח את הקובץ. כל כניסה בטבלה תצביע ל inode של אותו הקובץ.

3. **i-node table (of open files)** – מחזיקה מידע על הקובץ: מיקום, מתי ניגשו לאחרונה, גודל, מונה למספר הפעולות פתחו את הקובץ.

1. Search directory structure for the given file path
2. Copy inode into in-memory data structure (inode table)
3. Create an entry in system-wide open-file-table
4. Create an entry in PCB
5. Return a pointer to the location of the entry in the PCB to the user



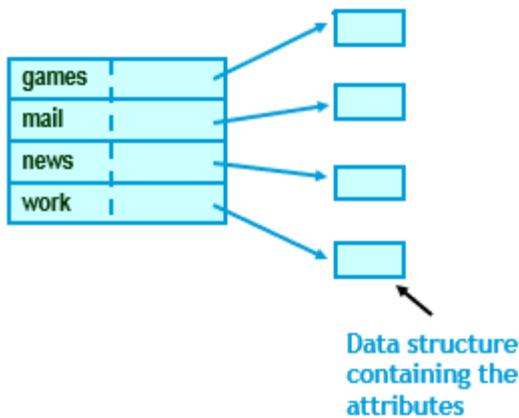
מימוש ספריות

שתי גישות למימוש:

1. יש קבצים מיוחדים, קבצי ספרייה, שבהם שומרים את המידע של הספרייה (מקומות בדיסק, הרשאות וכו').

games	attributes
mail	attributes
news	attributes
work	attributes

2. הספרייה היא קובץ אחד, קובץ טקסט, שמכיל רשימה של שמות של קבצים וכל אחד מהם ה-inode המתאים. (דוגמה לגישה במצגת)



Disk scheduling

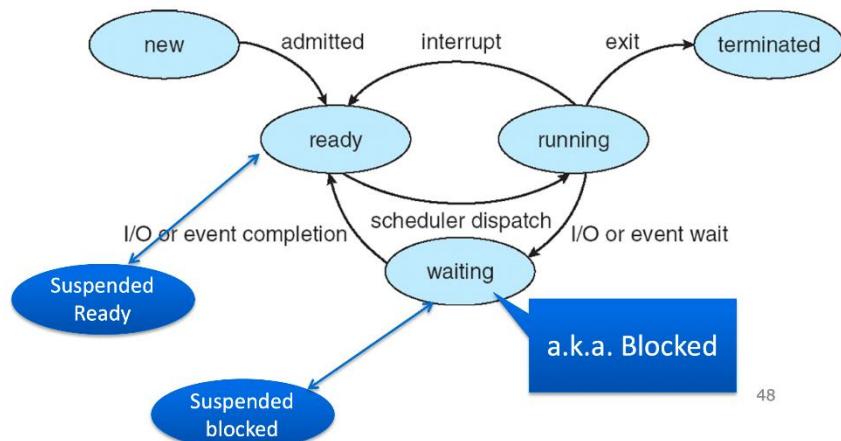
- בזמן הקריאה או הכתיבה לדיסק, יכולות להתבצע קריאות נוספות לדיסק. מערכת הפעלה שומרת קריאות לדיסק שעדיין לא הتمמשו והיא צריכה לבחור מי תהיה הקריאה הבאה. לשם כך יש את האלגוריתמים הבאים:
1. **FIFO**. אבל יכול לדרוש מצב שבו הזרוע המגנטי זהה הרבה כי הקריאה לא ניגשת למקומות קרובים בדיסק וזה מבזבז את ה-seek time שהוא נרசה למדוע.
 2. **Shortest seek time first (SSTF)** – בוחרים מבין הקריאה לדיסק בהתאם לזמן שהביי קרובה למקום בו הזרוע המגנטי נמצא בעתה. יכול לגרום להרעה.
 3. **Scan ("elevator algorithm")** – נועד למונע מצב של הרעה. מוחפשים בכךן אחד את הקריאה הביי קרובה.

Scheduling

תחכורת: ה-short-term scheduler איזה תהליך שנמצא במצב של ready יעבור למצב של running (scheduler, עובד מהר).

ה-high-priority scheduler – **Preemptive Scheduling** – יכול להפריע לתהליך שרגע ע"י הבנסה של תהליך אחר במקום running-ready.

הוסףנו שני מצבים תחיליים יכולם להימצא: **Suspended ready** ו-**suspended blocked**. נשעה חילוק מהתהליכיים שרצים כדי להגביל את הנסיבות. קיימים调度ers אחרים שאחראים על המעברים ועובדים בקצב יותר נמוך (mid-term scheduler).



48

יש תור של תהליכים – ג'ובים – שרצים את ה-CPU, וה-scheduler קובע מי מהם ישמש ב-CPU. dispatcher אחראי על ביצוע הפקודות של ה-scheduler – על context switch ועל מעבר ל-user mode.

סוגי ג'ובים

1. Long CPU burst – משתמשים בעיקר ב-CPU. מבצעים חישובים מורכבים ורחב לא צריכים I/O.
2. Short CPU burst – רוב הזמן משתמשים ב-I/O. scheduler צריך להתחשב בסוג הג'וב שבו הוא מירץ.

schedulers

- OS: CPU Scheduler (but also memory scheduler, and sometimes admission scheduler)
- Printer scheduler (or, in general, batch systems); schedulers for I/O devices (e.g. disk scheduler)
- Packet Schedulers in computer networks
- Scheduling request in a web-server

מדדים לטיב ה-scheduler

1. Throughput – מספר הג'ובים שהמעבד מצילח לסיים ביחידת זמן אחת. (# completed jobs) / (time unit)

אם המערכת יציבה throughput קבוע וזה שקול לו.
2. Turnaround time – כמה זמן ג'וב מחכה בתור עד שהוא מסיים (כולל waiting time). (running time + waiting time) / (# started jobs) (time unit)

משניות:

 3. Response Time – הזמן שלוקח ל-1 CPU burst הראשון להסתיים.
 4. Waiting time – הזמן בו ג'וב נמצא במצב של waiting.
 5. CPU Utilization – כמה זמן ה-CPU מבצע פקודות ייעילות?
 6. Fairness – הగנות בין הג'ובים השונים, שלא יורענו או יהיו בעליwaiting time גבוה.

לפעמים המטרות האלה סותרות אחת את השניה:

- נרצה למזער את throughput וכך להעלות את הקבוצה של כל ג'וב. אך עדיף לא לעשות context switch ארוך ורבעו response time.
- בכלל. אבל מצד שני, יהיה ג'וב עם burst ארוך ורבעו response time גבוהה.

בנוסף, נחלק את ה-schedulers לכמה סוגים:

1. offline vs. online – מסיקים על המחוור הבא לפי המחוור הקודם לעמודת online שאי אפשר לדעת מה יהיה.
2. Preemptive vs. non-preemptive – האם ה-scheduler יכול להפסיק ג'וב באמצע או לא?

- לא ברור אם ה-scheduler יודע על גודל הג'ובים. נצא בהתחלה מנקודת התחליה שכן.

FIFO

Job	CPU Burst
P1	24
P2	3

דוגמה: יש שלושה ג'ובים, ונניח שאין overhead על context switch. לכל אחד מהם יש CPU burst אחד, מעבר לזמן זה הם לא צריכים להשתמש ב-CPU. ה-scheduler יירץ את הג'ובים בסדר הבא:

Job	CPU Burst	Turnaround time	Waiting time
P1	24	24	0
P2	3	27	24
P3	3	30	27

$$\text{Average waiting time: } (0+24+27) / 3 = 17$$

$$\text{Throughput: } 3 / 30 = 0.1$$

אפקט השיריה: הרבה ג'ובים קצרים נתקעים אחרי ג'וב ארוך.

Shortest Job First (SJF)

כל פעם שבוחרים ג'וב חדש זה יהיה הג'וב עם הבן פחות time running.

בוגמה:

Job	CPU Burst	Turnaround time	Waiting time
P1	24	30	6
P2	3	3	0
P3	3	6	3

Average waiting time: $(0+3+6)/3=3$ Throughput: $3 / 30 = 0.1$

זמן ההמתנה הממוצע ירד, throughput לא השתנה.

האלגוריתם הוא offline, ידוע מה יהיה.

נעוד למחזר את average waiting time ועושה זאת

באופטימליות.

ניתן להשתמש באלגוריתם ב-preemption אם הג'ובים לא מגיעים בידך.

בוגמה:

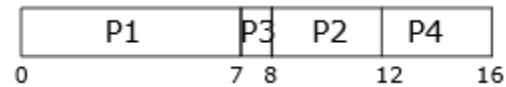
Job	Arrival time	CPU Burst
P1	0.0	7
P2	2.0	4
P3	4.0	1
P4	5.0	4

← ג'וב 1 מגיע ראשון. ה-scheduler רואה רק אותו ומריץ אותו. אם לא היה preemption, היה צריך לחכות 7 שניות לפני שיועשה פעילות אחרת.

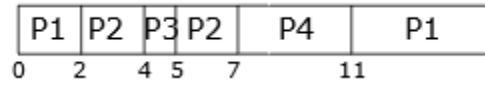
← ג'ובים 2, 3, 4 מגיעים. בזמן 7 ג'וב 3 יבחר.

← בזמן 8 הקלט הוא ג'ובים 2 ו-4. ג'וב 2 נבחר.

← בזמן 12 ג'וב 4 רץ.

Average waiting time: $(0+6+3+7) / 4 = 4$

Time	Scheduler Decision
0.0	Run P1
2.0	P1 has 5, P2 has 4. Preempt. Run P2.
4.0	P2 has 2, P3 has 1. Preempt. Run P3.
5.0	P3 finishes. Shortest remaining time P2. Run P2
7.0	P2 finishes. Shortest remaining time P4. Run P4.
11.0	Run P1.
16.0	P1 finishes.

אם יש preemptive scheduler-ים יכול בשmagim ג'וב חדש, להפסיק את הריצה של הג'וב הקודם ולהריץ את החדש. ה-scheduler יבחר להריץ את מי שנשאר לו הבן פחת זמן עד הסיום. **בוגמה:**Average waiting time: $(9+1+0+2)/4=3$

• האלגוריתם לא מקיים הוגנות.

~ את המשך ההריצאה אני אוסיף להרצאה הבא ~

הרצאה 13 – IO – Schedulers

Schedulers

Round Robin

ה-scheduler נועד לפתור את בעיות ההרעה והחוסר הוגנות שבאלגוריתמים הקודמים. לבסוף מוקצת יחידת זמן – quantum time, וכל ג'וב בתורו רץ על ה-CPU במשך יחידת הזמן זו. אם במשך הזמן הזה הג'וב לא סיים לרוח, ה-scheduler יחזיר אותו לסוף התור וירץ מישר אחר במקומו.

התורתנות:

- אלגוריתם הוגן – כל תהליך מקבל זמן מסוים על המעבד ובסיומו של דבר כל הג'ובים ירוצו.
- Waiting time מוגדר בזמן בו תהליך מחכה מרגע שנכנס לתור ועד שמקבל את ה-CPU לראשונה, ובאלגוריתם זה הזמן נמדד ותלוי בקונטנים ובמספר הג'ובים שמתינו לפניו.

דוגמאות:

Job	CPU Burst	Turnaround time	Waiting time
P1	20	30	10
P2	3	7	4
P3	7	18	11

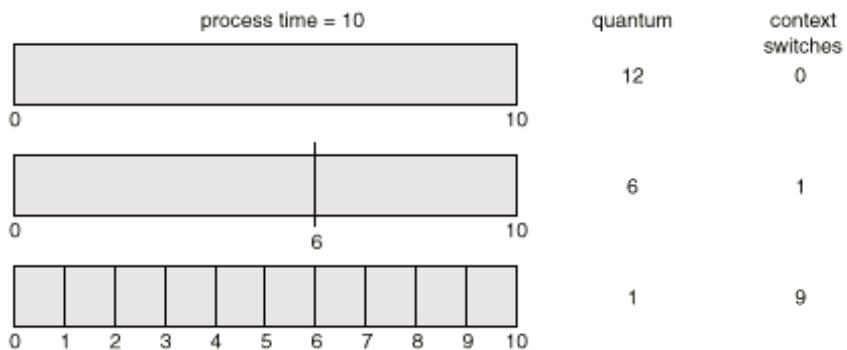


כל הג'ובים מגיעים באותו זמן context switch על overhead אין Quantum time = 4

באופן כללי, אם יש n ג'ובים ומוקצת q זמן קוןנטום, כל ג'וב מקבל $\frac{1}{n}$ מההריצה של ה-CPU במשך q זמן לכל היותר. אף ג'וב לא יჩקה יותר מ- $(1 - \frac{1}{n})q$ זמן בתור. גודל הג'וב לא משנה.

- ככל ש- q יותר גדול כך הפעולה תהיה דומה יותר ל-FCFS כי כל ג'וב יסימן את המשימה שלו לפני שיטות הקוןנטום.
- ככל ש- q קטן יותר כבها יש יותר context switches שגדילים את התקופה.
- עדיף שה-burst יהיה קצר יותר מאשר נמור מה- q כדי שהג'ובים יסימנו את הפעולה שלהם, וג'ובים נוספים יתור לא עכבות אוטם אבל גם מספר context switches יידך.

בגלובים אינטראקטיביים בעדיף קוןנטום נמור כי מה שחשוב הוא שיכלו לסימן את ה-burst הקצר בזמן התגובה $q(1 - \frac{1}{n})$ היותו נמור יותר טוב. קוןנטום גבוה מתאים ל-batching כי יש פחות context switch.

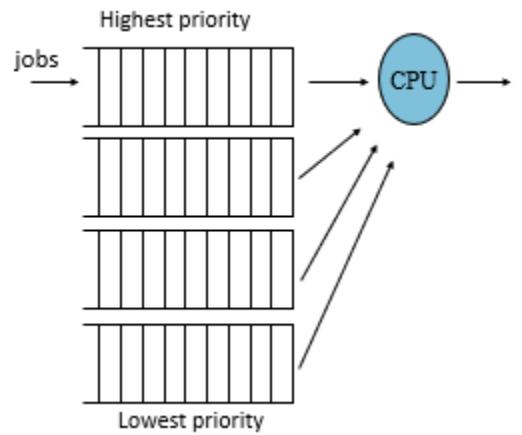


מימוש

בעצם ה-scheduler והתהליכים רצים על אותו CPU. ה-scheduler עצמו מתיישם להפסיק את הריצה של התהליכים כדי לוחץ בעצמו ויתכנן את החלטות שלן.

דרך אחת היא שה-scheduler יתנהג באופן non-preemptive ויחכה שתהליך רץ יפסיק בעצמו את הריצה וייתור על המעבד, וה-scheduler ייכנס בעצמו.

דרך נוספת ויתר מקובלת משתמשת בפסיקות שעון. גם כאן מוקצב זמן קוונטום כך שכל פעם שעובר הזמן זהה תיגרם פסיקת שעון שבה ה-scheduler ירוץ על המעבד ויחליט החלטות.



Priority Scheduling

במציאות לגיבים יש עדיפותות שונות ולכן יש schedulers שMRIIZIM את הגיבים לפי התיעודם שלהם. בד"כ יש מספר תורים וכל תור עדיפות מסויל, וה-scheduler קודם מרץ את כל הגיבים בתור של העדיפות הגבוהה ולאחר מכן יודע עדיפות עד לנמוכה ביותר.

- התורים יכולים להיות שונים אחד מהשני בדרך השימוש שלהם.
- SJF זה תיעודף לפי זמן ריצה.

הבעיה: הרעה.

הפטונות (בד"כ משולבים):

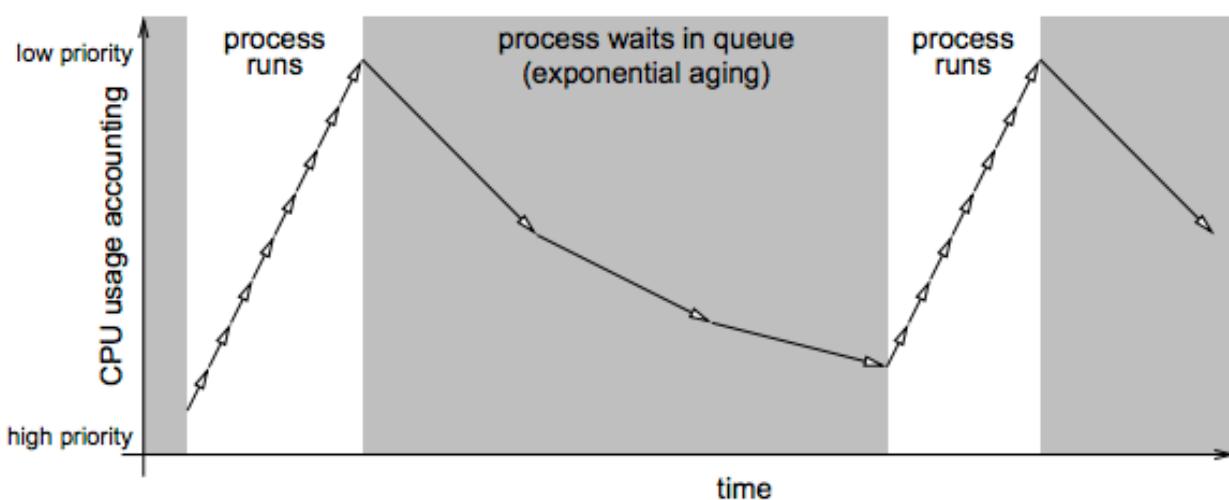
1. **Time Slicing** – הקזאה זמן CPU לתורים לפי התיעודם שלהם. לגיבים חשוביים יותר יהיה יותר זמן על ה-CPU והם יריצו מהר יותר אבל עדין נגיעה לפחות חשבים.



2. **Aging** – תיעודף לפי הזמן שהתהליך נמצא במערכת. שתי גישות לבר:

- ככל שהתהליך נמצא יותר זמן במהלך הפעלה שלו יודת והוא מקבל את ה-CPU פחות. ככל רוח התקנת הביצוע שלו תהיה מהירה אבל לקראת הסוף הביצוע יהיה יותר איטי. הגישה לא מנענת בהכרח הרעה (אלא אם כן ממומש גם time-slicing), ג'וב ארוך יכול להיות עז.
- ככל שהתהליך נמצא יותר זמן במהלך הפעלה שלו עולה.

במערכות ההפולה היסנות priority נקבע לפי זמן + בסיס מספרי שימוש לעדיפות הגבוהה ביותר. ככל שהמספר יותר גבוה כך הגיב פחות מתועד. זמן מעבד יורך פי 2 כאשר אין שימוש במעבד (דעיכה אקספוננציאלית לעומת עלייה לינארית).



Multilevel Feedback Queue

באשר יש כמה מעבדים, יש מערכת של כמה תורים עם עדיפותות שונות ווגבים יכולים לעבור בין התורים. בIMPLEMENTATION זה מגדירים:

- מספר התורים
- מימוש הזמן של כל תור
- מתי "לסדר" תחילך ומתי להפסיק
- באיזה תור יתחל תחילך שנכון

Real Time Scheduling

יש משימות שמקצת להן פרק זמן שבו הן חייבות להסתיים.

- חייבים לסיים באותו זמן – **Hard real time system**
- כראי לסיים באותו זמן – **Soft real time system**

סוגיSchedulers נוספים (שאפשר לשאול עליהם ב מבחן..)

- **Guaranteed scheduling**
 - for each process compute the ratio of actual CPU time consumed to CPU time entitled;
 - run the process with the lowest ratio.
- **Lottery scheduling**
 - give processes lottery tickets for various system resources, such as CPU time;
 - A lottery ticket is chosen at random, and the process holding the ticket gets the resource.
- **Fair share scheduling**
 - Each user/application gets the same amount of CPU time regardless of how many processes/threads it has started.
- **Numerous other possibilities exist!**

air לבחן scheduler

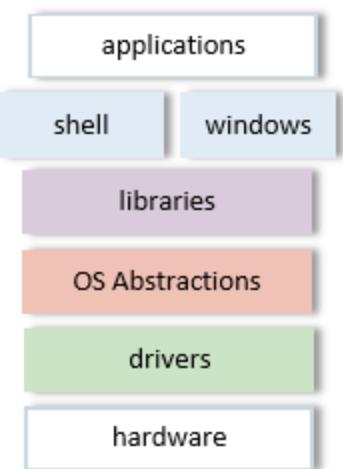
- **Deterministic modeling (Analytic evaluation)**
 - takes a particular predetermined workload and defines the performance of each algorithm for that workload.
- **Queuing models**
 - queuing theory, etc.
- **Simulation**
 - Simulations involve programming a model of a computer system.
 - A simulation is of limited accuracy
- **Implementation**
 - put the actual algorithm in a real system for evaluation under real operating conditions.

Input / Outputסוגי מכשיריםCommunication devices

- Input only (keyboard, mouse, joystick, light pen) –
- Output only (printer, visual display, voice synthesizers...) –
- Input/output (network card...) –

Storage devices

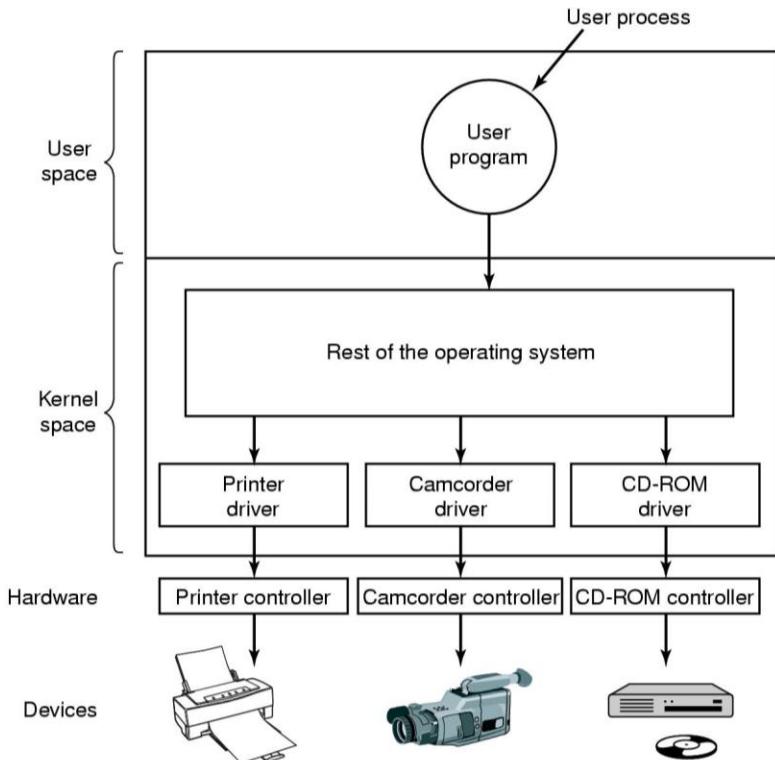
- Input/output (disk, tape, writeable optical disks...) –



Input only (CD-ROM...) –
Every device type/model is different •
input/output/both, block/char oriented, speed, errors, ... –

זיכרון: ה-*bus* מחבר את התקנים ומאפשר תקשורת ביניהם. אליו מחוברים גם ה-CPU וה-*memory*.
דיברנו גם על השכבות בר שמערכת הפעלה נמצאת בשכבה האמצעית ומהולכת לשכבות פנימיות. אחת השכבות היא השכבה של ה-*drivers* וນמצאת הći קרוב לחומרה.

- מה החלק של מערכת הפעלה בקשר להתקני קלט/פלט?
- אבסטרקציה של המבקרים כדי למנוע גילוי פרטיים קטנים וטיפול בשגיאות.
 - אפשר פעולה על המעבד במקביל לפעולות על ה-IO.
 - תמיכה בחלוקת ה-*device* לשיטופ של כמה תהליכים כולל הגנה וזמן בעט הצורך.



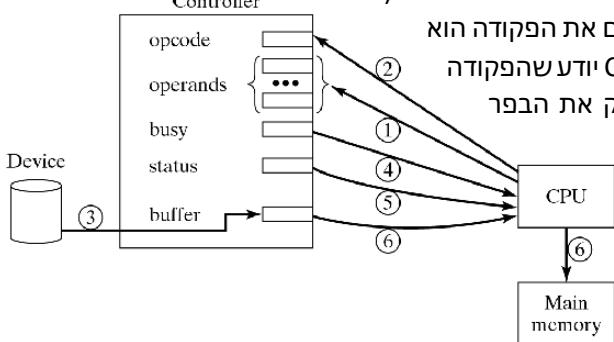
Device Controller & Driver

התקני הקלט פלט יש בחומרה עצמה יכולה לבצע – controller. כמו מעבד קטן שעושה דברים ספציפיים ועובד במקביל למעבד של המחשב. האינטראיס בין לין ה-*device-low* הוא controller יש במקביל אליו את ה-*drivers* שנמצאים במערכת הפעלה ואחראים על התקשרות עם ה- controller אליו הם מعتبرים פקודות. כאשר מסתiemת הפעולה הם אחראים להודיע על כך למערכת הפעלה. ה-API בין ה-*controller* לבין ה-*drivers* מאוד בסיסי: close, open, flush, seek, write, read. הקשר נעשה על גבי ה-*bus*, וה-CPU מבצע את הפקודות הרגילות שנרשומות ליכרן במקומם שמומפה לריגיסטרים החיצוניים.

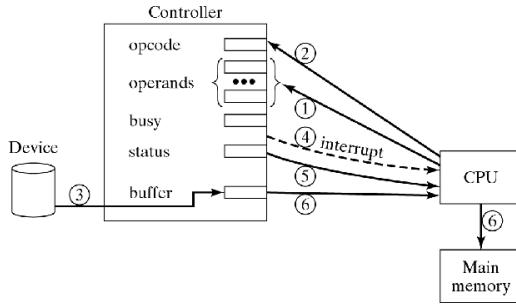
Polling vs Interrupting

יש שתי גישות להעברת מידע ממערכת הפעלה ל-*controller* ובזרה, ולהודיע על סיום הפעלה למערכת:

1. CPU – ה-*CPU* בודק متى מסיים. הוא יעביר את הפקודה הדורשת ל-*controller* (1-2).

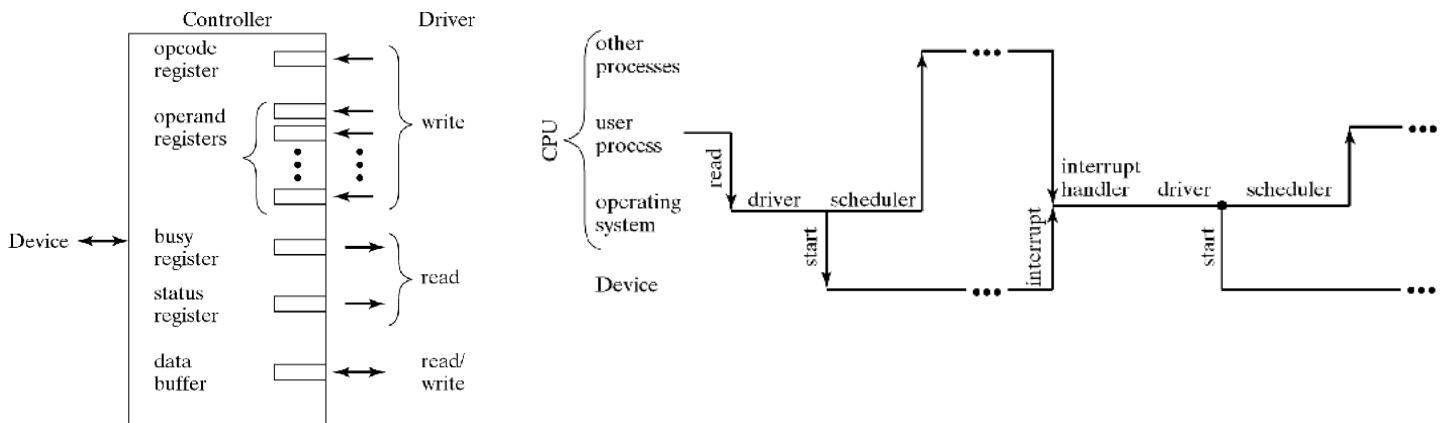


בשיטו – שיביר את הפקודה ל-device ← device ← בשה-device יסימם את הפקודה הוא כותב את הדטה לבפר (3) ← ויכבה את הביט (4) ← ה-*CPU* יודע שהפקודה הסתיימה (בודק בלולאה איןסוף אם ה-*busy* דלוק) ← מעתיק את הבפר לחיבורו הראשי. הבעה: .busy-wait.



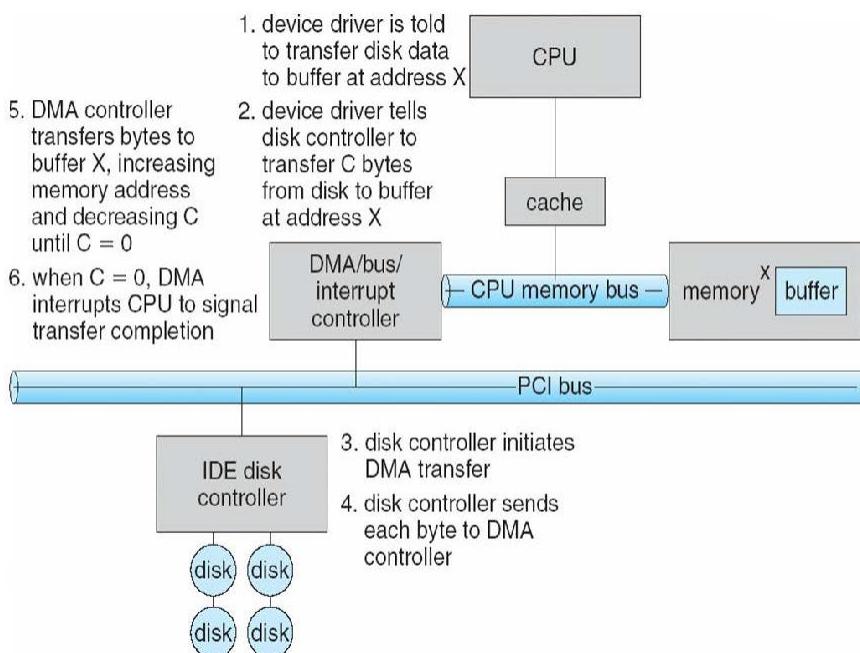
– ה-CPU אחראי על העברת המידע, אבל הוא ידע שהפעולה הסתימה ע"י פסיקה שיישלח ה-controller. ה-CPU יבודק את הסטוס ואם הוא מתאים יעתיק את הבפר ל ذיכרון הראשי.

ה-Workflow הבא מייצג איך הכל מתנהל:



Direct Memory Access (DMA)

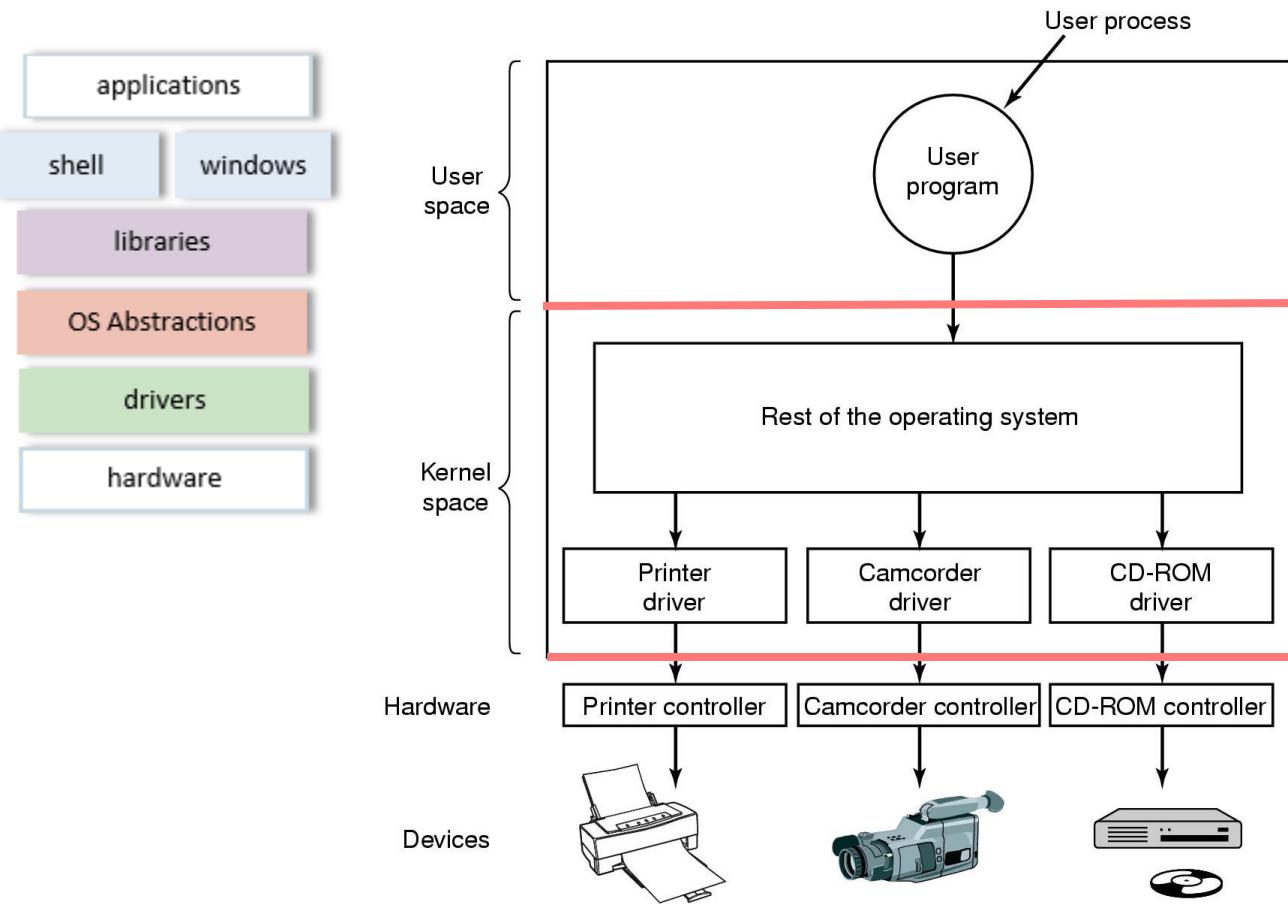
- האופן שבו הדטה עוברת מה.dev controller ל זיכרון הראשי במקומות הנדרש ועוד דורק את הפסיקה.
- גם על ה-**bus** יש **bus controller** כותב ל **main memory**.
 - ל **CPU** יש **controller** שמחובר ל **CPU** ול **DMA** מה שאומר שהם מתחברים על הגישה ל זיכרון.



הרצאה 14 – IO – 14

1 / 0

בשחסתכלנו על היררכיית מערכת הפעלה, בצד התיכון הקרוב לחומרה נמצאים ה-drivers. ה-drivers מותקנים בתוך מערכת הפעלה ורצים כחלק منها. הם מקיימים ממashing עם ה-I/O, בחומרה יש את ה-controllers שמתקשרים עם ה-drivers.



Device Independent Technique

מה שבפועל קורה הוא שגם ה-drivers מאוגדים לקבוצות כך שה-I/O של drivers בקבוצה אחת מתנהגים זהה:
devices – Block-oriented •
devices – Stream-oriented •
devices – Network-oriented •

כדי לא למשך כל פעם את אותן פעולות משותפות ל-devices מסוימת קבוצה, במערכת הפעלה יש רכיב שדואג לניהול devices מסוית סוג. למשל מערכת הקבצים היא האינטראקטיבית לניהול **block-oriented**, **stream-oriented**, **Network-oriented**。

ה-**stream-oriented** device independent technique נתן מענה במקרים הבאים:

- Buffering
- Error Handling
- Device Scheduling/Sharing

Virtualization

Virtualize vb (computer science) (tr) to transform (something) into an artificial computer-generated version of itself which functions as if it were real.

ביצוע פעולה ממשיות ע"י מזוינה.

דוגמאות:

1. **חלוקת האוד דרייב למחיות**, כל מהיצה חשבת שכט הדיסק שלו.

2. **Virtual Memory**

3. **Virtual Private Network (VPN)** – רשת בין מחשבים שנבדית ברשות פרטית אבל היא בעצם רשת ציבורית.

Virtual Machine

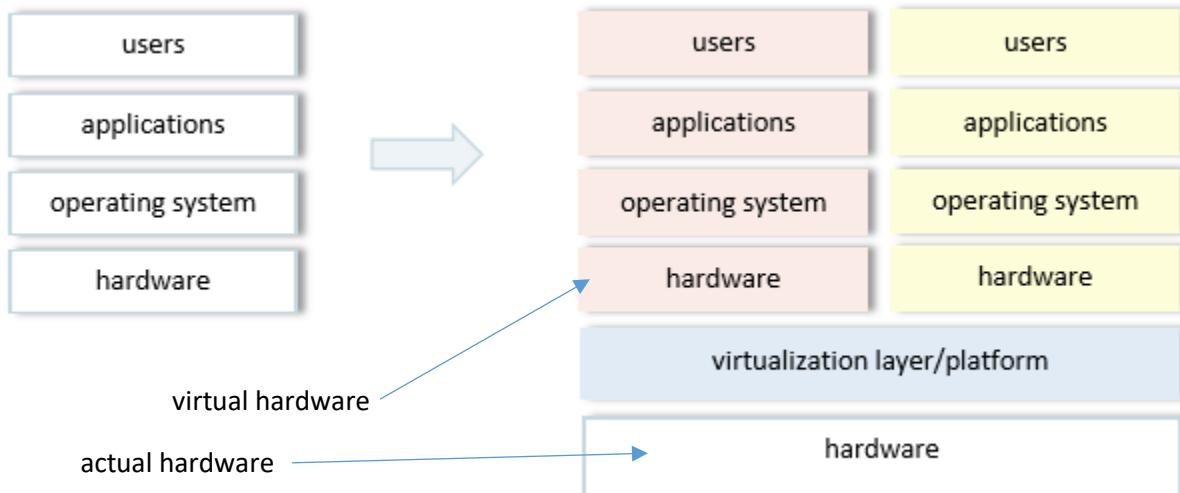
"A VM is an efficient, isolated duplicated of a real machine"

המכונה הוירטואלית מקיימת:

- **Duplicate** – מכונה מזוינה שמתנהגת כמו מכונה אמיתי. המכונה תכלול את כל מה שיש במכונה אמיתי, אך האפליקציה שרצה ב-VM לא יודעת להה שהיא רצתה בסביבה וירטואלית.
- **Isolated** – אם יש כמה VM על אותו שרת אף אחד לא יודע על קיום الآخر, כמובן כל VM רץ על מכונה אחת בלבד. אם ירצו לתקשר אחד עם השני יעשו זאת ע"י socket.
- **Efficient** – מהירות הריצה של המכונה המזוינה צריכה להיות זהה לשול האמיתית.

Virtualization Layer

נרצה להריץ ב-VM ביחיד החומרה האמיתית (CPU, זיכרון וכו'). הפלטפורמה זו צrica תמייהה של המעבד. בשביל זה יש שכבה בשם **virtualization layer/platform/hypervisor** שהיא מתווכת בין החומרה האמיתית לבין ה-VMs.



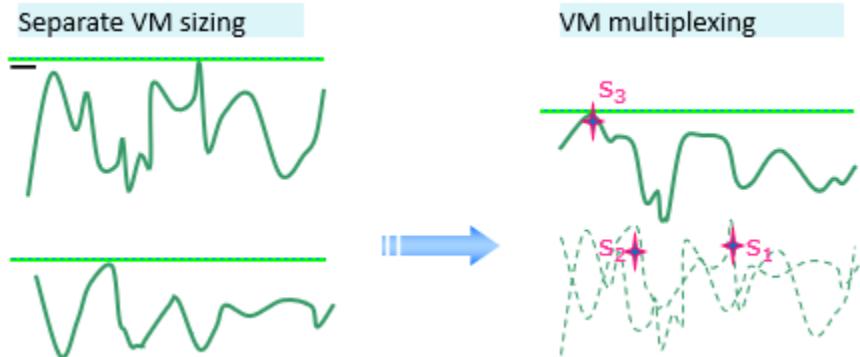
– החומרה של ה-VM מתנהגת כמו חומרה רגילה ומימוש CPU, memory, I/O, etc.
האפליקציהים ומערכות הפעלה. שכבת הוירטואלייזציה מקשרת בין החומרה הוירטואלית לבין האמיתית.
– שכבת הוירטואלייזציה מספקת אבטוחה ומיופי תוכנה שרצה ב-VM לתוכנה
שרצה במחשב אמיתי.

הסיבות לשימוש ב-VM

1. **Server Consolidation** – רוצים לצמצם את כמות servers ולבודד ביניהם כך שלא ישפיעו אחד על השני.
 - לחוב ה-servers יכולים להריץ רק אפליקציה אחת, כך שאם יש הרבה servers CPU usage שלהם נמוך מאוד.
 - הריצת כמה servers על VM חוסן בחשמל ובאנרגייה (שכל server בפני עצמו צריך).
2. **Disaster Recovery** – אם server נופל, ב-VM קל לגרום לו לשוב, לעובד במקום אחר ולהתואושש מהבעיה.
3. **High Availability** – תמיד ניתן יהיה הגיע ל-server (בגל הסיבה הקודמת).
4. **Testing and Deployment** – לעיתים רוצים לבצע פעולות בסביבות קטנה ולא בחומרה האמיתית שיכולה להתקלקל/להיבבות וכו'. אם משאנו רע קורה ב-VM זה לא משפיע על החומרה האמיתית.
5. **Desktop Consolidation** – תמיiba ב-legacy applications, אפליקציות שרוכזות על מערכות הפעלה ישנות.

VM Workload Multiplexing

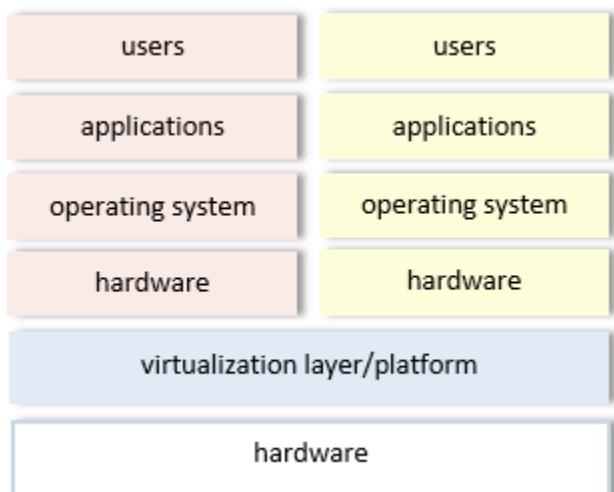
בצד שמאל בשרטוט יש גרפף שמייצג עומס של שני VMs רצים של servers שונים, ובצד ימין גרפף הריצה של ה-VM על אותו ה-server, עם סימון של זמן הפסגה. ניתן לראות שכשMRIIZIM על שרת אחד זמן המקסימום S קטן יותר לעומת הזמן שבס-VM צריכים את המחשבים הוא לא דואק חוף, וכשהמחרבים אותם מצליחים לחשון.



VM Encapsulation

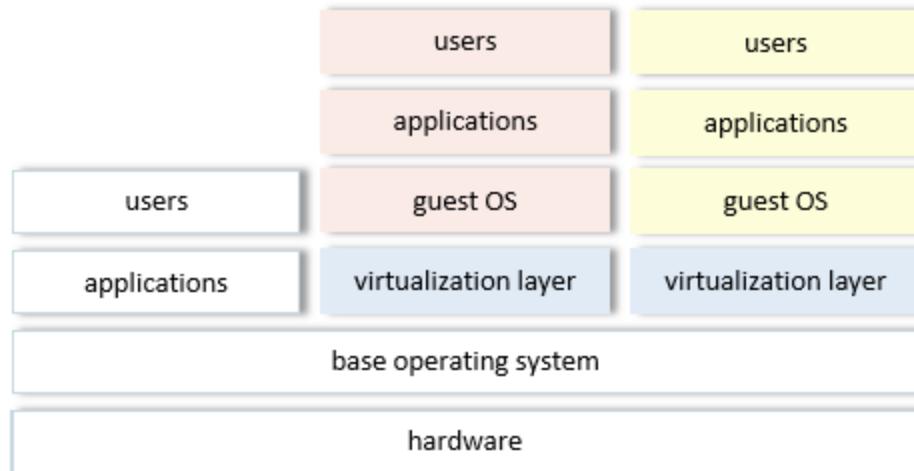
VM encapsulation נוגע במימוש ה-VM. ה-VM רצים בתוכנה – גם הזיכרון, גם האפליקציות, גם ה-CPU וכו'. את מצב התוכנה הנוכחי ניתן לשמר בקובץ ולהשתמש בו. בקובץ יהיה את כל הדטה, מצב הזיכרון וה-devices ועוד. את אותו הקובץ יוכל להריץ על חומרה אחרת או לשכפל אותו.

- כדי לעדכן את הקובץ נצטרך לעשות snapshot, לשמר את המצב הנוכחי.
- העברת הקובץ נקראת migration.
- הפעלת המערכת תהיה מהירה מאוד.

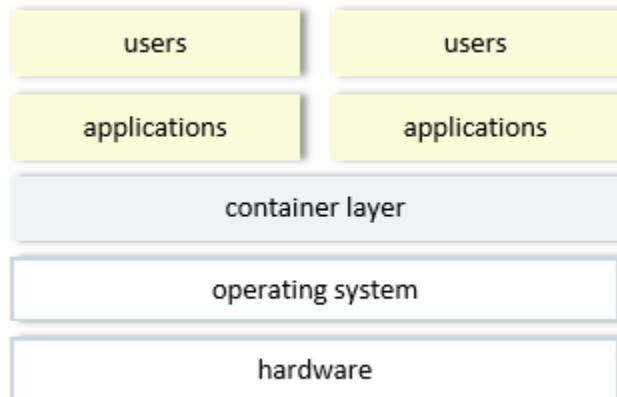


סוגי hypervisors (שכבות הווירטואלייזציה)
1. **Type 1 Hypervisor – Native**. מעל החומרה האמיתית רצה שכבת הווירטואלייזציה שמריצה VMs.

.2. שבבת הווירטואלייזציה רצהCAF aplikatsia בתוך מערכת הפעלה ה"מארחת",
מערכת הפעלה האמיתית של המחשב. מערכת הפעלה מרצהCAF aplikatsia נוספת בילי קשר ל-VM.



.3. שבבת הווירטואלייזציה היא בעצם ה-OS-level – Container / Docker הפעלה וירטואלית. בעצם מערכת הפעלה מקבלת קונטינר שמכיל כמה תהליכים ומרצהCAF אותם.



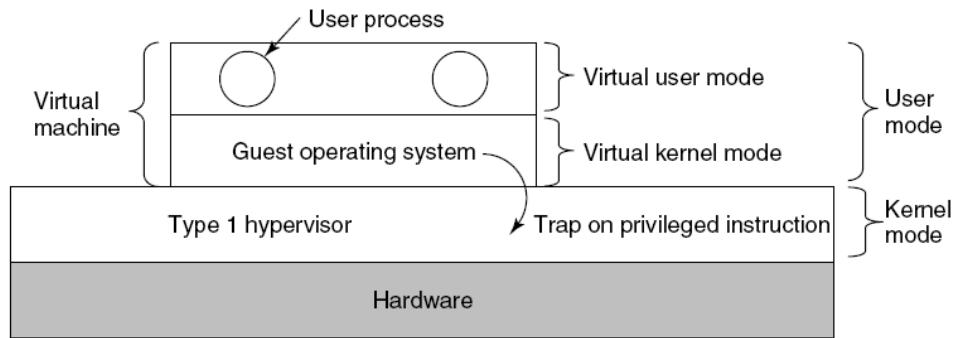
Virtualization Mechanics

ישן במה שיטות מרכזיות לתמיכה בוירטואלייזציה עליה נubby.

Trap and Emulate

ץ על ה-1-type. את ה-VMs מרים ב-user mode לא תיוקם מערכת הפעלה. לא רצה לחת הרשות גישה ב-kernel mode ל-VM שנכתב ע"י מישחו חיצוני, כדי להגן על מערכת הפעלה האמיתית. ה-OS של ה-VM רצה ב-user mode, אבל חושבת שהיא רצה ב-kernel mode kernel Amiyti.

בשה-OS הווירטואלית תנסה לבצע \leftarrow privileged instruction \leftarrow החומרה תראה שההוראות הטענו ב-kernel mode ותזרוק שגיאה \leftarrow השגיאה תיתפס ע"י ה-hypervisor (ה-hypervisor יכול לעבוד ב-kernel mode)

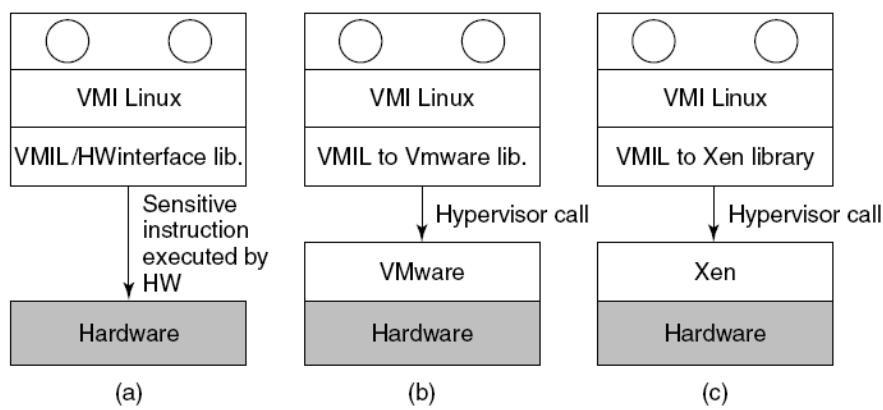
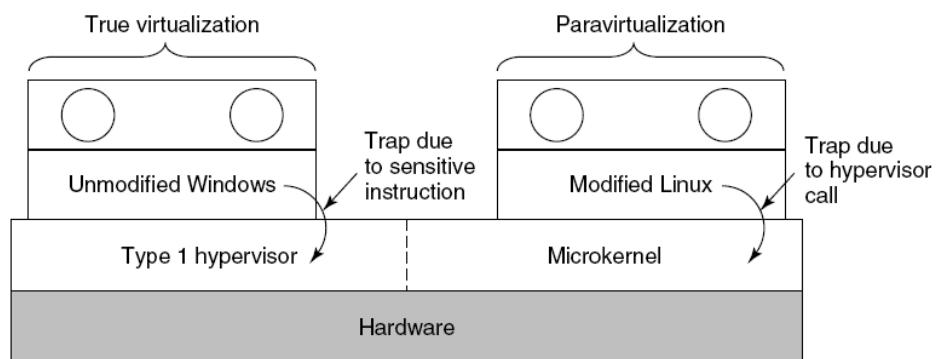


Binary Translation

לפי שיטה זו, פקודות ב-user mode רצות בלי בעיה על ה-CPU הפיזי. אבל פקודות ב-virtual kernel mode יעברו קודם לhypervisor שבודק אם הפקודה היא חוקית ובסדר או שהיא "שונה" ו"מיוחדת" (משמעותה שונה בין ה-user לבין ה-kernel). אם היא בסדר היא תורץ, אם היא לא "בסדר" היא מתורגמת למקום מבוצעות סט אחר של פקודות במקומם ותורף כדי מנסים לתרגם לפקודות הבינאריות הנכונות ע"י trap בך שיהיה אפשר לבצע(type 1).trap and emulate (ב-2).hypervisor type רץ במערכת הפעלה והוא משבטב את הקוד כל פעם שנתקל בפקודה אם מדובר ב-2.hypervisor type, הוא משבטב את הטרנסלטוריון בזאת ולא יהיה צריך לבצע שוב את התרגום.

Paravirtualization

מערכת הפעלה של ה-VM מدع שהיא נמצאת בסביבה וירטואלית, וכך היא תדע לא לקרוא למערכת הפעלה האמיתית אלא ל-hypervisor. הרבה יותר קל ונוח מהשיטות הקודמות כי הוא כולל רק שינוי הקוד של מערכת הפעלה ב-VM. הבעיה: כל פעם שרוצים מעלה חומרה/מערכת הפעלה אחרת צריך לעשות שינוי חדש.



Hardware Assistance

ה-CPU מודע לכך שיש הרצה וירטואלית. בנוסף ל-user mode ו-kernel mode יש מצב נוסף, מצב ביןים, והוראות שנתמכות ע"י החומרה (הרצת VM, סגירת VM). יש מבנה נתונים נוסף – VM control block – VM control block.

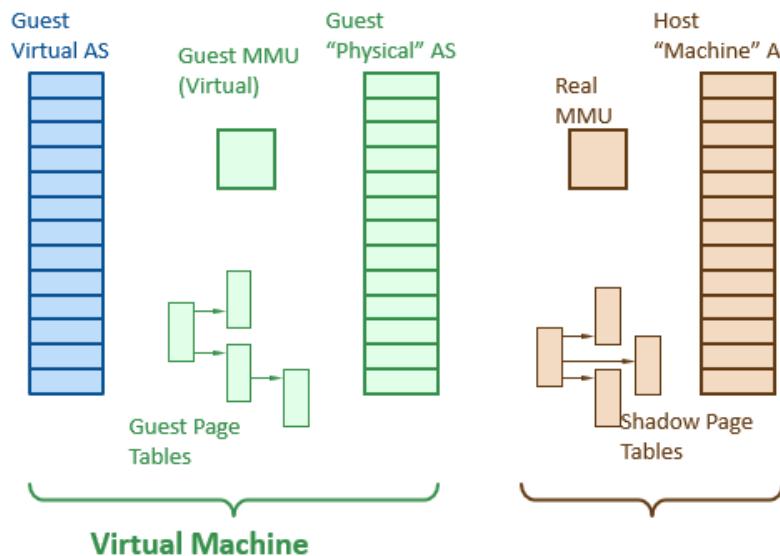
Virtual Memory

במקום ה-MMU יש VMMU שמקבל בתובת וירטואלית ומחייב guest physical address, בתובת שלא יושבת בזיכרון הפיזי ואיתה יהיה צריך למפות לזכרון הפיזי.

ישן כמה שיטות מיפוי, ונדבר על אחת בשם Shadow Page Table. בשיטה זו, מרחיב הכתובות הוירטואליות ממופה ל-guest physical AS. שם המיפוי לזכרון האמתי עבר דרך guest shadow tables שה-OS לא מודע לקוימן.

שאלות שעולות:

- איך שומרים על עקבות בין ה-walk לguest shadow ?
 - איך מעדכנים את TLB במעבר בין VMs ?
- שאלות שעולות:-paravirtualization ותמייה של החומרה יכולות לפתור את השאלות.



I/O Virtualization

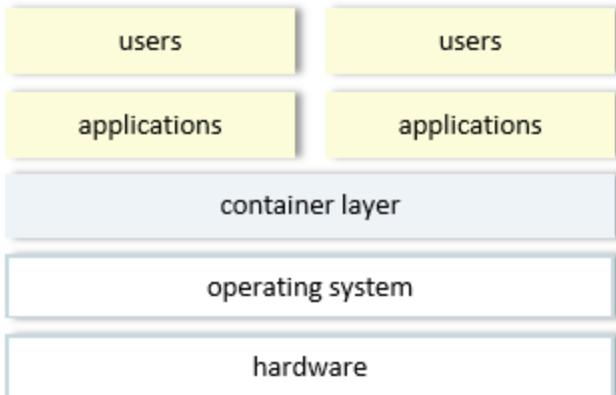
ב-VM ה-drivers נמצאים בתוך ה-OS guest וה-controllers controllersアイテム הם צריכים לתקשר עם אמיטיים וצריך לתווך ביניהם, וזה ע"י ה-I/O virtualization / שונעה ע"י 3 שיטות מרכזיות:

1. **אמולציה** – כמו trap and emulate, הדרייבר מוציא איזושהי פקודה, ה-OS מקבל אותה ועשה איזושהי אמולציה כדי להוציא אותה לפועל.
2. **Split** – וירטואלי driver עם device driver ברמת ה-device driver, הוא יודע שהוא לא אמיתי.
3. **Pass-through** – בעיקר ברשותה תקשורת בשיש צורך בהעברת DATA בכמות גדולה. התקנת driver אמיתי ב-VM יידע להעביר שירות מידע מה-device ל-VM. הhypervisor יש גישה ישירה לחלק מהתקני הקלט פלט.

Containers

יש מערכת הפעלה אחת המשותפת לכל הקונטיינרים, מעליה שכבה בשם container layer שמעליה רצים כל הקונטיינרים שמכילים אפליקציות ויזרים. בעצם עוטפים כמה תהליכים קשורים בקונטינר אותו מרים מעל מערכת הפעלה וזה חוסך התקינה של סביבת העבודה.

מעין וירטואלייזציה למערכת הפעלה. לכל קונטינר יש זיכרון, מערכת קבצים ומשאבים של מערכת הפעלה עצמו. בעצם הקונטיינרים הם שלב ביןים בין תהליכים לבין ה-VM, למשל תהליכים משתפים מערכת קבצים והקונטיינרים לא.



- הكونטינרים מכילים בעיקר:
- Application
 - Dependencies
 - Libraries
 - Binaries
 - Configuration files

Containers vs. VM

- VM מכילה בתוכה מערכת הפעלה מלאה אך ברוב המקרים אין צורך בכר ומספיק להשתמש בקונטינר שיתור קל לשימוש.
- בקונטינרים משתמשים בכל מה שמערכת הפעלה מספקת וכן מתקשרים עם החומרה, לעומת VM שעבודת עם שכבת ה-hypervisor. لكن הקונטינרים עובדים מהר יותר.
- מאותה סיבה של מעלה, מכך שה-VM שעבדת עם ה-hypervisor היא יותר מבודדת ובטוחה, משתפים פחות.
- VM יכולת לזרז על מערכות הפעלה שונות מאותו host והקונטינרים רצים על אותה מערכת הפעלה (host).
- קונטינרים הם וירטואלייזציה של מערכת הפעלה ו-VM של החומרה.

יתרונות של קונטינרים

- קטנים
- פחות שימוש במשאבים
- קל ליצור
- ריצה מהירה יותר

חסרונות של קונטינרים

- פחות בידוד ופחות ביטחון
- פחות גמישות, יותר ספציפיים למערכת הפעלה אליה יועדו

 **בצלחה!**