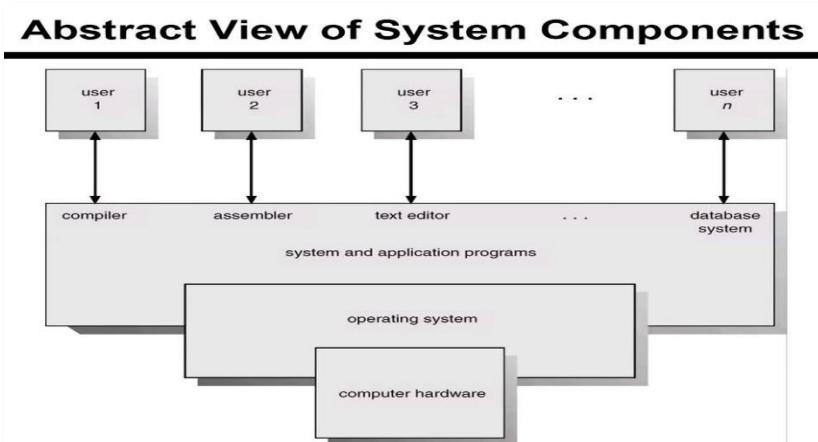


# קורס מערכות הפעלה

## הרצאה 1

- מהי מערכת הפעלה? – סוג של תוכנית שמתווכת בין המשתמש למחשב.
- מטרת מערכת הפעלה? – להריץ תוכניות משתמש, להשתמש מחומרה בצורה יעילה.
- מה יש במערכת מחשב? –

- Central Processing Unit – **CPU** ○
- העברה של נתונים פנימה או החוצה, קלט מהמקלדת, פלט למסך, קלט או פלט מהדיסק וכו'. ○
- Application programs** ○  
נתונים וכו'.
- Users** ○  
משתמשים (בני אדם, מכונות, מחשבים אחרים) וכו'.

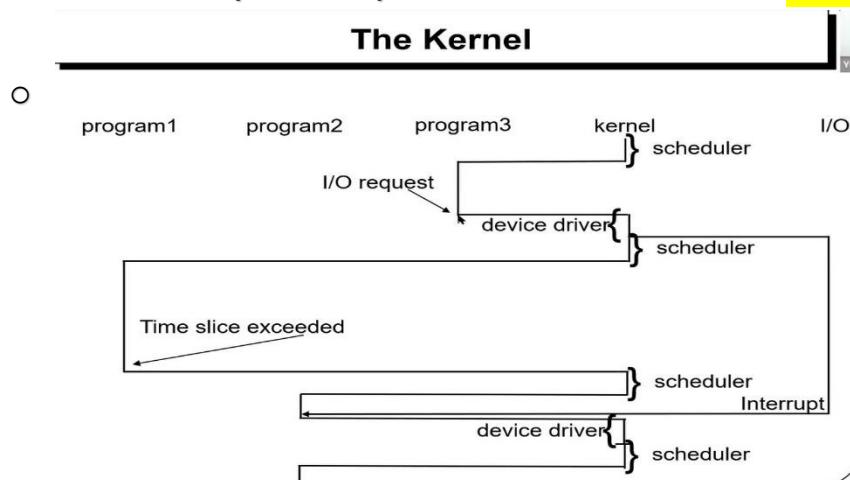


- **הגדרת מערכת הפעלה** – שולטת בהוצאה לפועל של תוכניות, תוכניות יבקשו מממשק המערכת בקשה ומערכת הפעלה תחליט האם ומתי לתת לתוכנית לבצע את מה שהיא רוצה לעשות, בנוסף מנהלת את התקני הפלט קלט (O/I).
- **Kernel** – הגרעין של מערכת הפעלה, קבועה מה ירצה, על איזה ליבת, איך יהיו הפקצות זיכרון, איזה הרשות יש לכל תוכנית וכו'.

- **Multiprogramming Systems** – כמה תוכנית רצوت כביכול בו בזמןית, אך לא באמת, מערכת הפעלה מקצה לכל תוכנית זמן מסוים שהען האנושית לא מסוגלת להבחן בו וכך ריצה בין תוכניות, כך היא מדמה פעולה במקביל של התוכניות. חלק מהזיכרון מוקצה למערכת הפעלה חלק לתוכנית אחרת וחלק לתוכנית אחרת וכו'.

נכתב על ידי אוחד אمسلم, תשפ"ב.

- מחליט איזה תוכנית להריץ וכמה זמן הפעלה תנתן לכל תוכנית. **Scheduler**



- בקשה להביא קובץ מהדיסק עוברת למערכת הפעלה
- חТИכת קוד במערכת הפעלה שאחראי על טיפול בבקשת הפעלה של התקן חומרה מסוים, למשל דיסק, והוא מבקש מהחומרה להביא לו את הבלוק הרלוונטי בזיכרון, דבר שלוקח זמן.
- מפסיק ריצה של תוכנית לאחר X זמן כדי לא לbezבץ משאבים.
- פסיקה – "פסיקה" מייצרת אירוע חומרתי שגורם למעבד לקפוץ לקוד של מערכת הפעלה, מערכת הפעלה מבינה שהייתה פסיקה, בודקת באזור הזיכרון הרלוונטי למה הייתה פסיקה ומבינה שהມידע מוכן והתוכנית מוכנה לפעול.
- מה צריכה לדעת מערכת הפעלה? – כל מה הקשור ב- O/I, ניהול זיכרון, להחליט מי התוכנית הבאה שתறוץ.
- אזרזumi בדיסק שבו נשמרות פקודות שלוקחות הרבה זמן עד שהפעולה מתבצעת, למשל תוכנית שمبיאה לשלוח קובץ להדפסה. **Spooling**
- מיליה נוספת ל- **Time sharing systems**, Multiprogramming Systems – מערכת ההפעלה מבצעת המון תוכניות שנשמרות בזיכרון ונניח שאין עוד מספיק מקום בזיכרון RAM, אז מערכת הפעלה תבחר תוכנית מסוימת ותזרוק אותה באופן זמני לדיסק, פעולה זו נקראת "Swap File" וכן יכול כבירוק לשחרר מקום בזיכרון.
- **Parallel Systems** – מערכות מקבליות, כיום רוב המחשבים הם מקבליים, ככלומר בעלי כמיה ליבות, אך בעצם תוכניות באמת יכולות לפעול במקביל ולא רק לדמות מקבליות בעזרת קופיצה מהירה מתוכנית לתוכנית. (ליבת מילה נרדפת למעבד).

בריבוי ליבות רגילה, הליבות חולקות אותו זיכרון ואותו שעון.

- **למה לעבוד בריבוי ליבות ?** – בשביל תפוקה מוגברת, כלומר יותר פעולות מחשב ביחידת זמן אחת.
- **עיבוד מקבילי סימטרי** – כל המעבדים הם שווים מעמד, כלומר אין מעבד שמנוהל את כל המעבדים, ככלם רואים את אותה תמונה זיכרון.
- **עיבוד מקבילי אסימטרי** – מצב שבו יש מעבד אחד שהתפקיד שלו הוא לנוהל את כל שאר המעבדים, כלומר תפקידו להקנות לכל שאר המעבדים את העבודה.
- **מערכת מבוזרת** – מערכת שבזורת את כוחות העיבוד, במקומות להשתמש במחשב על, היא משתמשת בהרבה מאוד מחשבים רגילים שמחולקת את העבודה, הם מחוברים בראש תקשורת והם משתפים פעולה אי-ఈשו. (למשל כר גוגל עובדים).

#### יתרונות מערכת מבוזרת:

- אמינות.
- חלוקה של משאבי.
- אפשר לעשות הרבה הרבה עבודה בו זמן.

#### Real-Time Systems

- מערכת ש מגיבה לאירוע מסוים בזמן אמיתי, למשל ניהוג של טיל, צילום ווידאו, המערכת צריכה להתמודד עם הנסיבות הנוכחיים בקצב שבו הם נוכנסים.
- **Hard Real-Time** – מערכות שצריכות להיות מוכחות שעובדות במגבילות זמן ספציפיות, לא יעבודמערכות שצריך לחלוק איטם את הזיכרון, בספק אם יהיה בה מערכת הפעלה ואם תהיה אז היא תהיה מאוד מוגבלת.
- **Soft Real-Time** – צריך שהמערכת תגיב בזמןים הרצויים, למשל אם לצורך צפייה בסרטון יוטיוב יש דרישת מעבר של נתוני מהשרת של יוטיוב דרך הרשת, לפרק את הדחיסה של סוג הקובץ, לשולח לכרטיס מסך את התמונות בקצב של הפריים הספציפי, לשולח את הנסיבות של הקול לפסקול וכל זה צריך לקורות ב 1/30 שניות או שהוא כזה, אבל אין הבטחה מתמטית שהסרטון ירוז רצוף ולא נפספס פרייםים אבל זה

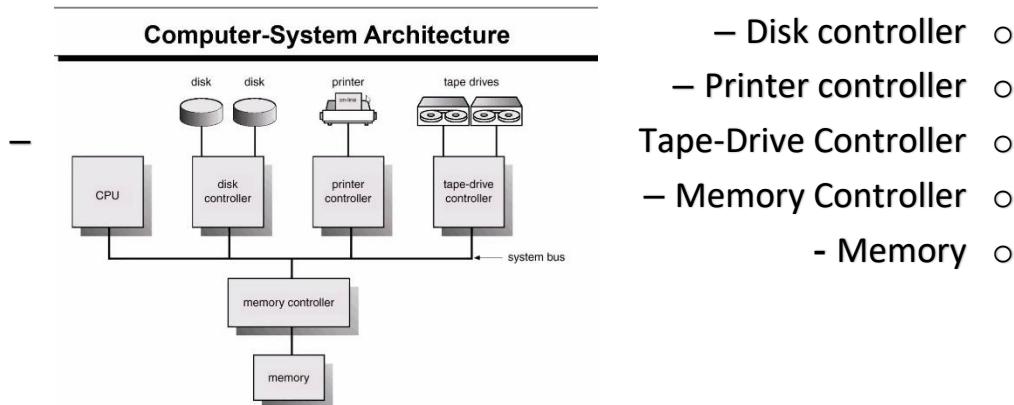
בסדר כי אנחנו ערוכים לזה שנайд פריטים מדי פעם ולכז זה נקרא Soft Real-Time.

## Real-Time Systems

- Hard real-time:
  - guaranteed worst-case response times.
  - Secondary storage limited or absent, data stored in short term memory, or read-only memory (ROM)
  - Conflicts with time-sharing systems, not supported by general-purpose operating systems.
- Soft real-time
  - A soft real-time system is one in which its users are happier when the system responds optimally, but which is not considered to have "failed" when the system doesn't meet each and every desired response time.
  - Useful in applications (multimedia, virtual reality) requiring advanced operating-system features.

## הרצאה 2

- ארכיטקטורה של מחשב מלפני 30 שנה
  - עצם הקו עליו רץ החשמל מה CPU לכל הרכיבים האחרים.



- לכל בקר חומרה(**Controller**) קיים באפר מקומי, וכל אחד יש ISR.
- ה- **Device Controller** מודיע ל**CPU** על איזושהי פסיקה עצלה.
- **(ISR) (Interrupt service routine)** – אחראית להחזיר את האוגרים למצב הקודם שלהם כדי שהתוכנית משתמש לא תרגיש שהייתה הפרעה.
- ארכיטקטורת הפסיקות חייבת לשמר את כל הכתובות של הפסיקות.

נכתב על ידי אוחד אمسلم, תשפ"ב.

- הפסיקות נשמרות בווקטור פסיקות שמכיל את כל הכתובות של הפקנציות במערכת הפעלה.
- במהלך טיפול בפסקה, פסיקות שנכנסות מנוטרלוות כדי למנוע מצב של איבוד פסיקה נכנסת, כלומר יש טיפול בפסקה אחת בכל פעם.
- **A Trap** – פסיקה שנוצרה בעקבות פעילות משתמש או בעקבות אישזה **Error**.
- מערכת הפעלה מונעת על ידי פסיקות.
- מערכת הפעלה שומרת את המצב של ה**CPU** ידי שמירת הרשומות של ה-

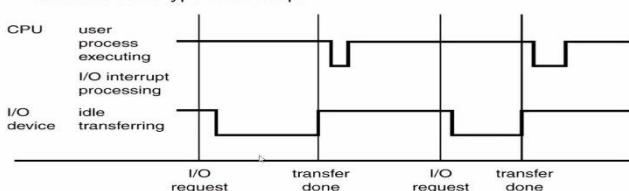
## Program Counter

---

### Interrupt Handling

---

- The operating system preserves the state of the CPU by storing registers and the program counter.
- Separate segments of code determine what action should be taken for each type of interrupt



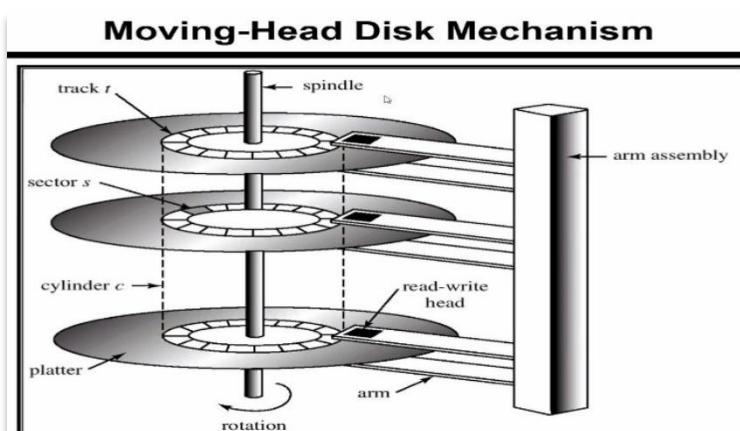
- **DMA (Direct Memory Access) Structure** – אפשרות שבו בקר זיכרון מסויימים מדברים עם הזיכרון (כלומר כתבים לזכרון), ללא התערבות של ה-CPU, זה מאפשר לשחרר את ה-CPU לטיפול בבית, שימוש בהתקנים שכותבים הרבה מידע, דיסקים, כרטיסי רשת, כרטיסי מסך וכו'.
  - **Storage Structure**
  - **Main memory** – זיכרון ראשי, הוא נדייף, ככל אם נכבה את המחשב אז הוא יעלם.
  - **Secondary storage** – כל מה שהוא זיכרון לטוווח ארוך, ככלומר ניתן להגעה למידע מתי שנרצה.
  - **Magnetic disks** – בקר הדיסק אחראי לדעת לאיפה כתובים, כמה מסילות יש בדיסק וכו'.
- למשל דיסק קשיח, הדיסק הקשיח מורכב מדיסקות שטוחות העשוויות מאלומיניום או זכוכית, מצופות חומר מגנטי ומחוברות יחדיו לציר אחד. בין הדיסקים מציה טבעת-רווח צרה החובקת את הציר ומטרתה לשמר על מרחק קבוע ביניהם. לכל צד של דיסקה צמוד **ראש קריאה** וכתיבה המותקן על זרוע. במהלך פעולה הדיסק, הדיסקים מסתובבют יחד ואילו הראש נע הלאר וחזרה כדי לאפשר לו להגיע לכל נקודה ונקיודה על פני הדיסקה. כלל הראשים נעים ייחודי, כך שראש אחד נמצא על

נכתב על ידי אוחד אمسلم, תשפ"ב.

גלאי מסויים על גבי הדיסקה, כל הראשים נמצאים על גלאי זה בשאר הדיסקות.

ראש הקריאה והכתיבה מכיל למעשה שני רכיבים נפרדים, אחד לקריאה ואחד לכתיבה. רכיבים אלו קרובים מאוד אחד לשני והם עשויים סליל מתכת זעיר בגודל של פחות ממיילימטר רכיב הכתיבה מוחול שדה מגנטי ובכך מוגנתת את שכבת הציפוי של הדיסקה במהלך הסיבוב.

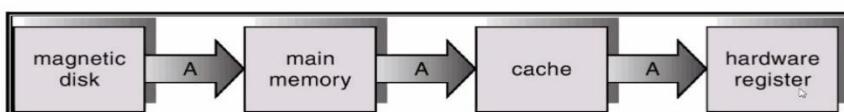
בזמן הקריאה מושררת השדה המגנטי שנכתב חזרה בסליל רכיב הקריאה. הנזונים נשמרים בשפה ביןארית משומש כל שדה מגנטי זעיר מסמן "0" או "1" בהתאם לכיוונו.



מסמן "0"  
או "1"  
בהתאם  
לכיוון.

## Storage Hierarchy •

### Migration of A From Disk to Register

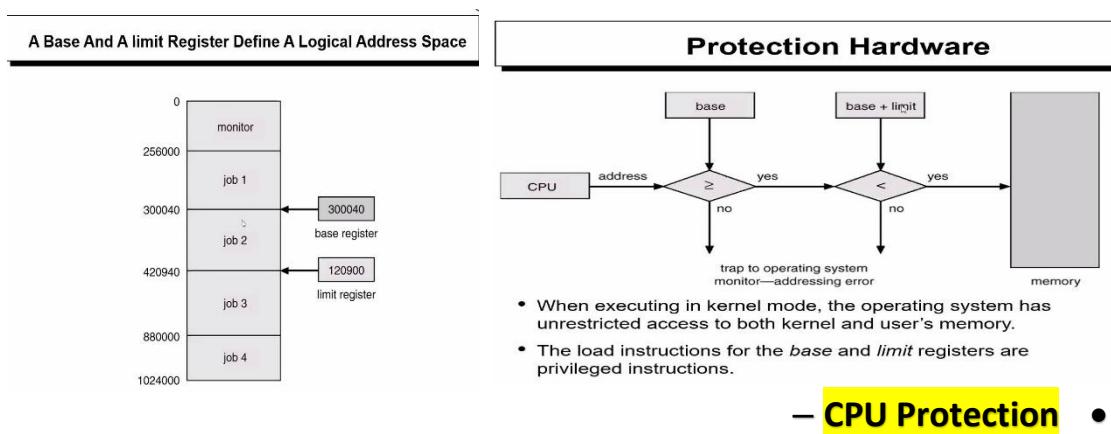


- Use of high-speed memory to hold recently-accessed data.
- Requires a *cache management* policy. Caching introduces another level in storage hierarchy. This requires data that is simultaneously stored in more than one level to be *consistent*.

**Dual-Mode Operation** – בעצם מגן על התוכניות במערכת הפעלה, קלומר לתוכניות אסור לגשת שירותים להתקני חומרה, ופעולות של תוכנית מסוימת לא יכולה להפריע לפעולות של תוכנית אחרת, המעבד צריך לדעת בכל פקודה

האם זו פעולה משתמש או פעולה של מערכת הפעלה. (User mode/Kernel mode).

- המעבר בין שתי ה Modes מתבצע באמצעות פקודה Set user mode ו-interrupt/Fault. כאשר זה מתבצע באמצעות שינוי ביט מ- 1 ל- 0, יש פקודות מסוימות שנitin לגשת אליהם רק באמצעות User mode. הפקודות הם בעיקר גישה לזיכרון וגישה לתקני I/O מה שלא ניתן לעשות ב User Mode.
- System Call – השיטה בה תוכנית מסוימת מבקשת מערכת הפעלה בקשה מסוימת, הבקשה מתורגמת לפסיקה, ההבדל בין פסיקה זו לפסיקות אחרות היא שזו פסיקה תוכניתית יוזמה, גורם למעבר ל- Kernel mode.
- Memory Protection – אוגרים מיוחדים שרק מערכת הפעלה יכולה לגשת אליהם, הגנה בה תוכניות משתמש לא יכולים להרשות את הטבלה של ה interrupt service routine, הגנה מפני קריאה וכתיבה.
- Base Register – מחזיר את הכתובת של תחילת הזיכרון של התוכנית שרצה כרגע.
- Limit register – מכיל את גודל הטווח (מספר הבטים בטווח).
- כל הזיכרון שמחוץ לטווח זה מגן.
- אם יש זליגה לכתובות שלא כפי שהוגדר אז יהיה Error.



○ Timer – מערכת הפעלה קובעת אחרי כל כמה זמן הוא מאותחל, לאחר שהטיימר פוקע הוא מייצר פסיקה שמעבירה את השליטה חזקה למערכת הפעלה, הוא מקפיץ את ה- Program Counter להתחלה של ה - ISR (interrupt Service Routine) שמתפל ב- Timer interrupts. ספציפי ומונ הסתם מי שמתפל בפסיקה של טימר זה ה- scheduler.

- השימוש בטיימר הוא רכיב קרייטי בכל מערכת המשמשת ב-Time Sharing, כלומר הוא בעיקר עבור חלוקת זמן בין תוכניות.
- ניתן להשתמש בטיימר גם כדי לחשב את הזמן שעון קיר הנוכחי.
- שליטה בטיימר היא פעולה **Privileged instruction** כלומר רק מערכת הפעלה יכולה לשלוט בו.

### הרצאה 3

- **Process Concept** – תהליך, תוכנית שרצה ויש לה איזשהו הקצאת זיכרון שבו יש את הקטע קוד שלו.
- **אזורים הזיכרון של תהליך הן כליהן:**
  - האזור שבו יש את הפקודות מכונה של התוכנית.
  - האזור בו שמורים המשתנים, למשל מה שהקצתנו בעזרה `Malloc`.
  - המחסנית בו הפרמטרים של הפונקציה שמורים.
- **Process Creation** – תהליך נוצר כי תהליך אחר יצר אותו, ומה נוצר שיש עץ של תהליכיים.
- **Resource Sharing (משאבים משותפים):**
  - **כל תהליך יש תוכנית פתוחה** – בוינדואם קבצים פתוחים לא משותפים בין אב לבן.
  - **בלינוקס האב והבן מתחלקים בין קבצים פתוחים**, כלומר הבן יכול להשתמש בתהליך שהבא פתח.
  - באיזשהו שלב התהליך בן יסתהים, התהליך של האב ישולף את הערך המוחזר, כלומר האב קורא את הערך זהה ומתקבל אינדיקטיה על האם הבן הצליח לעשות את מה שהוא צריך או לא.
  - **מרחיב זיכרון** – בלינוקס הבן הוא העתק של התהליך של האב, בוינדואם נוצר תהליך חדש ונטען אליו איזשהו exe. חדש.

#### **Process Creation**

---

- Parent process creates children processes, which, in turn create other processes, forming a tree of processes.
- Resource sharing
  - Parent and children share all resources.
  - Children share subset of parent's resources.
  - Parent and child share no resources.
- Execution
  - Parent and children execute concurrently.
  - Parent waits until children terminate.
- Address space
  - Child duplicate of parent.
  - Child has a program loaded into it.

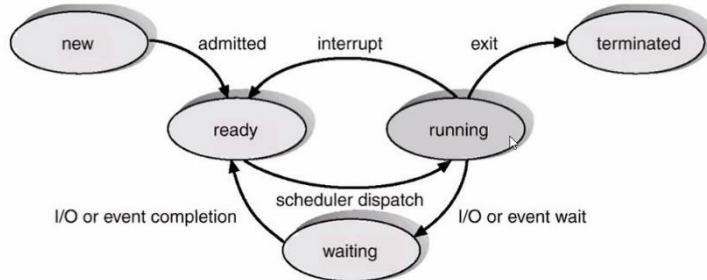
## • Process Termination (מתי תהליך מסתיים):

- מסתיים כשהוא יוצא מהבלוק או שיצאה פקודת Exit.
- Wait ממתין שתהליך בן יסתים ומחזירה האם התהליך בן הסטים באופן נורמלי או באמצעות פקודת Kill.
- כאשר רצחו אותו בעזרת פקודת Kill.
- בxit אם תהליך מסתיים אז כל התהליכים שמתחתיו בעץ מאומצים על ידי ה- **init**, הערך המוחזר מאותו תהליך יוחזר ל- **xit** והוא יזרוק אותו לפח.
- **init** - זה התהליך הראשון, כלומר תפקידו הוא להריץ את כל התוכניות הבאות אחרים, ה- **init** מופעל על ידי ה- **Kernel**.
- יש מערכות הפעלה שהורגות את התהליכים של הבנים כאשר תהליך האב מסתיים.

## • Process State – מצבים של תהליכיים באופן כללי:

- **יצירת תהליך** - יוצרים תהליך חדש.
- כאשר המתזמן (**Scheduler**) מחליט שהתוכנית תרוץ אז התהליך נמצא נמצא על המעבד וMRIIZ פקודות, ברגע שהייתה פסיקת שעון אז הוא יכול להיות במצב של **Ready** כלומר התהליך מוכן לרוץ ורק מחייב שהמתזמן יקצת לו זמן לרוץ.
- **Waiting** – ממתין שהוא יקרה, למשל אם התהליך בקש לקרוא משוח מהדיסק אז הוא ממתין עד שה-**O/I** הזה מסתיים, או שההתהליך בקש להמתין כי הוא בקש שהוא מהרשת וכו'.
- **Ready** - התהליך מוכן לרוץ ורק מחייב שהמתזמן יקצת לו זמן לרוץ.
- **Terminated** - כשההתהליך מסתיים אז הוא במצב **Terminated** ואז בסופה של דבר הוא ימחק מהטבלה של התהליכים.

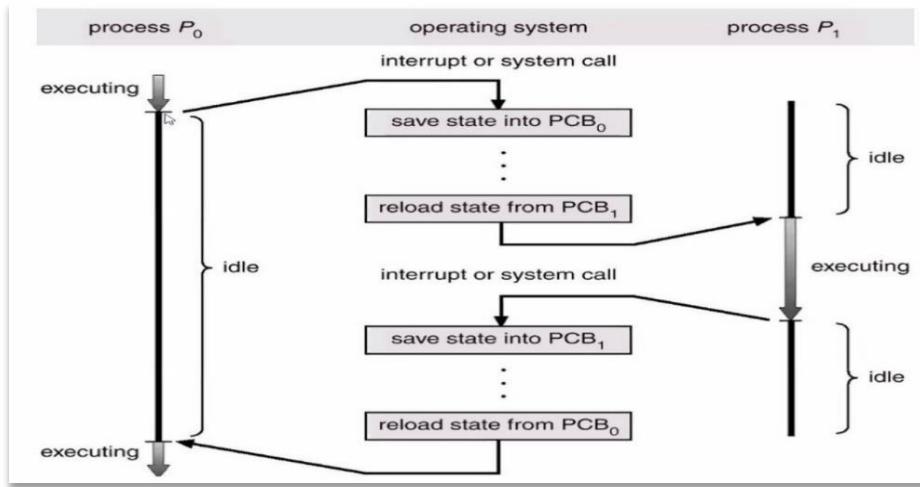
## - Diagram of process state ♦



### ♦ process state in Linux

- R - מצב שמאפשר גם את Running וגם את Sleep.
- S - מצב שבו כלומר קלוםר.
- T - תהליך שהוקפא, כלומר Stopped.
- Z - תהליך זומבי.
- D - Wait - O/I.
- **תהליך זומבי** – תהליך שהסתיים, כלומר קרא ל- Exit למשל אבל האב עדין לא קרא את הערך המוחזר.
- **טבלה שבה מופיעים כל התהליכים PCB (Process Control Block)** – כל כניסה בטבלה התהליכים מכילה מבנה נתונים, כלומר חתיכת אינפורמציה שמייצג Process.
- ♦ מערכות הפעלה צריכים לדעת על כל Process:
  - מה הוא Process State של התהליכי.
  - מה הוא המספר סידורי של התהליכי.
  - מה הוא Program Counter של התהליכי.
  - מה הוא CPU register של התהליכי.
  - מה הוא CPU scheduling information של התהליכי.
  - מה הוא memory management information של התהליכי.
  - מה הוא Accounting information של התהליכי.
  - מה הוא I/O information (טבלה של קבצים פתוחים).

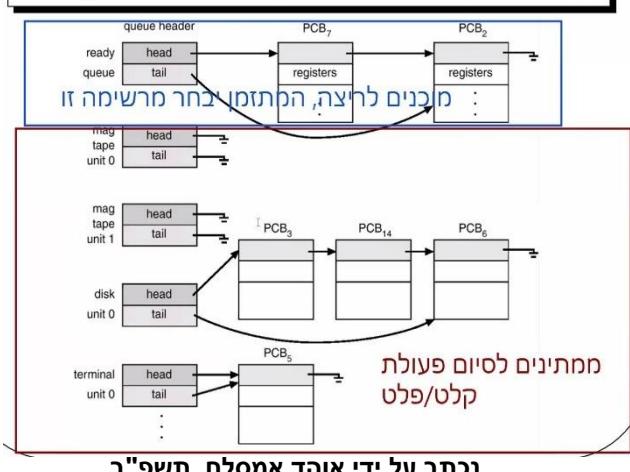
## -CPU Switch from Process to Process •



## - Process Scheduling Queues •

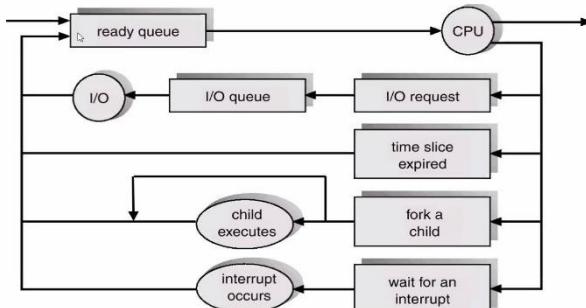
- **אוסף התהליכים שモכנים לרוץ, ה- scheduler צריך** לבחור תהליך מבין ה- **Ready queue**.
- עברו כל התקן חומרה יש אוסף של תהליכי **Device queue** שמתינו עבור התקן O/I מסוים, תהליך שנמצא ב-**queue** לא יכול להיות ב- **Ready queue** בו זמינות כי התקן ממתין למשהו, למשל לקריאת משחו מהדיסק ולכון הוא לא **Ready**.
- **התהליך כל הזמן מטייל בין התורים**, כלומר חלק מהזמן הוא נמצא ב- **Ready queue** כאשר הוא מוכן לרוץ, וחלק מהזמן הוא נמצא ב- **Device queue** של הדיסק כי הוא ממתין לנזtones של הקובץ או שהוא נמצא ב- **queue** של המקלדת כי הוא עשה `Scarf` ורמשתמש עוד לא סיים להקליד וכו'.

### Ready Queue And Various I/O Device Queues



נכתב על ידי אוחד אמסלם, תשפ"ב.

### Representation of Process Scheduling



- **Long Term Scheduler** – פעולה המתבצע במחשב על מנת פרויימים קיימים מתזמן ארוך טווח שקבע את הרמה של כמה תהליכיים ירצו במקביל, משמש עבור תהליכיים שיכולים לרצות תקופה ארוכה מאוד.
- במערכות מודרניות לא **קיים** **Long Term Scheduler**.
- **תהליך באופן כללי הוא CPU-bound או O-bound:**
  - **CPU-bound** – תהליך שמלה את רוב הזמן בלחכות ל-**O/I** יהיה מוגדר **O-bound**.
  - **CPU-bound** – תוכנית שפועלה זה להכפיל מטריצה של מיליון על מיליון אז רוב הפעולה תרצוץ ב- CPU והוא תהיה מוגדרת **CPU-bound**.
- **Context Switch** – כאשר ה- CPU מחליף בין תוכניות אז הוא צריך לשמר את המצב של התוכנית הקודמת ולטען את המצב של התוכנית החדשה. **-Context Switch** יש תקורה של זמן, ככלمر במהלך החלפה בין התוכניות המערכת לא עושה שום עבודה עילית. כדי לצמצם את זמן התקורה ניתן לתת לכל תהליך יותר זמן לרוץ ברכף אך המערכת תהיה פחות רטפונסיבית.
- **Cooperating Processes** – תהליכיים יכולים לשיתף פעולה אחד עם השני, למשל שני תהליכיים יצרו קשר כלשהו אחד עם השני והם יכולים לשיתף מידע אחד עם השני.
- **Producer-Consumer-Shared-Memory Solution Model** – ישנו שני תהליכיים אשר ה- **Producer** מייצר איזשהו יחידת עבודה ודוחף אותה למקום מסוים בזיכרון וה- **Consumer** לוקח אותה לפיקד. ניתן למשתמש במודול בעזרתו זיכרון משותף. ניתן לחשב על בapr מעגלי שה- **Producer** מכניס אליו את היחידות ואז ה-

### Producer-Consumer - Shared-Memory Solution

- *Producer process produces information that is consumed by a Consumer process. How to implement?*

- Shared data

```
var n;
type item = ... ;
var buffer. array [0..n-1] of item;
in, out: 0..n-1;
```

- Producer process

```
repeat
...
produce an item in nextp
...
while in+1 mod n = out do no-op;
    buffer [in] :=nextp;
    in :=in+1 mod n;
until false;
```

busy waiting



Consumer needs to implement the logic to take items from the buffer.

The producer needs to implement the logic to produce items into the buffer.

The buffer itself is shared memory, so both processes can access it simultaneously.

This pattern is called **Busy Waiting**.

Written by myself, Amos, Tsfiv.

הקוד עבר ה- Consumer הוא ד"י זהה, יש לו מגבלה של יכולת למלא רק 1-ח מהבאפר.

בקשרות זו מדובר על תקשורת בין שני תהליכיים. **Direct Communication** – תקשורת ישירה, תקשורת יותר מתחכמת כמו TCP,

**Indirect Communication** – יכול לתמוך בתקשורת בין הרבה תהליכיים להרבה תהליכיים, ניתן לחשב על זה כעל לוח הודעות משותף.

### **:Synchronization** •

- - כאשר אמרים Blocking איז זה אומר שהתהליך צריך להמתין עד שהפעולה מתבצעת במלואה, למשל פקודות Scanf, scanf מוגדר סינכרוני.
- – שגר ושכח, שולח הודעה וממשיר לבצע פקודות מהצד מקבל וישלוף אותו בזמןו החופשי, כלומר אני לא ממתין עד שזה יקרה, -Non-Blocking מוגדר אסינכרוני.
- – הם בעצם קריאות מערכת (System Calls) שמאפשרות תקשורת גם ברשת תקשורת IP/TCP וגם תקשורת בין תהליכיים והם יכולים להיות מוגדרים במידה Non-Blocking או Blocking.

### **:Buffering** •

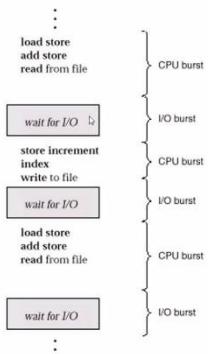


## **Buffering**

- Queue of messages attached to the link; implemented in one of three ways.
  1. Zero capacity – 0 bytes sender always blocks - waits  
for receiver  
Sender must wait for receiver (rendezvous).
  2. Bounded capacity – finite length of  $n$  bytes  
Sender must wait if the link is full.
  3. Unbounded capacity – infinite length  
Sender never waits.
- In UNIX there are:
  - Pipes – Bounded capacity
  - Messages – Unbounded capacity

## הרצאה 4

### Alternating Sequence of CPU And I/O Bursts



### CPU Scheduling •

- על ידי זה שאנו מרכיבים כמה תוכניות במקביל אנחנו מנוטים להגיא לניצול מקסימלי של ה- CPU.
- תהליך מסוים עושה חישובים ואז עושה I/O, זה מידול פשוטי של תהליך.
- CPU Burst – פרק הזמן שבו תוכנית מבצעת פקודות במעבד באופן רציף, כלומר עושה חישובים אריתמטיים, קוראת מהזיכרון כותבת לזכרון וכו', כלומר כל מה שלא קשור ל-I/O.

- תפקידיו של ה-Scheduler הוא לבחור בין התהליכים שמוכנים לרוץ ב-queueReady בין התהליכים שאינם תקועים ב-I/O ולהקצות להם זמן ריצה עד שמשהו יקטע את הפעולה.
- ה-Scheduler מחליטמתי לבצע החלטה כלומר לתת לתהליך זמן לפעול כאשר:

1. כאשר התהליך עובר ממצב של Waiting למצב של Running, כלומר במאור במאור בו התהליך ביקש לבצע פעולה I/O.
2. כאשר התהליך עובר ממצב של Ready State למצב של Running, כלומר בתפקידו הוא פועל, בעוד של התהליכים שמוכנים לרוץ,متى זה קורה ? למשל במצב בו התרחשה פסיקת שעון, כלומר נגמר הזמן שהוקצה לתהליך לרוץ.
3. כאשר התהליך עובר ממצב של Waiting למצב של ready, כלומר, כלומר פעולה I/O הסתיימה.
4. כאשר התהליך במצב של Terminates, כלומר כאשר התהליך הסתיים, למשל exit().

נכתב על ידי אוחד אمسلم, תשפ"ב.

## מספר 1 ו- 4 נקראים **Nonpreemptive**, 2 ו- 3 נקראים

### .Preemptive

- **Non-preemptive** – הzbתעה פסיקה יזומה של התהילר כמו (`exit`) חלק מהפעולות של התהילר.
- **Preemptive** – רצף הפקודות של התהילר נקטע בעקבות אירוע חיצוני באופן לא חזוי.
- **Dispatcher** – הוא חלק מה- Scheduler, ניתן לחשב עליו כקבילן ביצוע שלו, כלומר אם ה- Scheduler צריך לזמן את זמן הפעולה של כל תוכנית אז ה- Dispatcher צריך לדאוג להכין לו את הקרקע لها. למשל אם תהילר X צריך לרוץ אז ה- Dispatcher צריך לארגן את המצביע הנוכחי באוגר המוועדת של ניהול זיכרון שנדבר עליו בהמשך, לשולף מה- PCB את הערכי אוגרים הרלוונטיים כדי שהטהילר הרצוי ימשיך לרוץ מהנקודה הרצiosa שלו, להעביר את המערכת ל- User Mode ולבסוף לבצע את הקפיצה בתוכנית לנקודה שבה עצרנו,コレmur ל- Program Counter השמור מה- PCBコレmur לנקודה בה התהילר עצם בפעם האחרונה.
- לסיכום: ה- Dispatcher הוא חתיכת קוד בתוך ה- Scheduler אשר עשויה לפעול את כל התהילר זהה, היא אינה אחראית על קבלת החלטה אלא על הביצוע.
- **Dispatch Latency** – כמה זמן זה מبذבז,コレmur הזמן המתוקורה של כמה זמן לוקח לעשות את כל התהילר של ה- Dispatcher.
- **Scheduling Criteria** – איך נחליט איזה תהילר יבחר? המטריה היא לנצל את החומרה שלנו כמה שיותר.
- **Throughput** – לספר כמה תהילכים מתחילה ומסתיים במינימום ביחידת זמן, אנחנו רוצים שהמספר יהיה גדול,コレmur כמה שיותר תהילכים שישתינו ביחידת זמן אחת.
- **Turnaround Time** – מדידת הזמן שלוקח להריץ תהילר מסוים מרגע התחלה עד לסיומו.

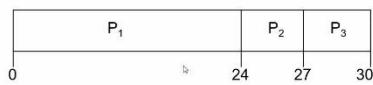
- Ready – למדוד כמה זמן התהליך ממתין ב- Waiting time queue (נשאף בזמן מינימאלי).
- Response Time – למדוד את הזמן שעובר מהרגע שמתחילים להריצת תהליך עד הפלט הראשון למשל הדפסה הראשונה.
- : FCFS First-Come First-Served Scheduling
- מдинיות scheduling בה תהליך שmagiu ראשון מקבל זמן פעולה ראשון, מתחילה ועד סוףו ללא הפרעות, הפעולות מופעלות לפי סדר הגעתן.

#### First-Come, First-Served (FCFS) Scheduling

- Example:

Process	Burst Time
$P_1$	24
$P_2$	3
$P_3$	3

- Suppose that the processes arrive in the order:  $P_1, P_2, P_3$ . The Gantt Chart for the schedule is:



- Waiting time for  $P_1 = 0; P_2 = 24; P_3 = 27$
- Average waiting time:  $(0 + 24 + 27)/3 = 17$

#### FCFS Scheduling (Cont.)

Suppose that the processes arrive in the order  $P_2, P_3, P_1$ .

- The Gantt Chart for the schedule is:



- Waiting time for  $P_1 = 6; P_2 = 0; P_3 = 3$
- Average waiting time:  $(6 + 0 + 3)/3 = 3$
- Much better than previous case.
- Convoy effect short process behind long process



- (אפקט השיריה) Convoy Effect – יש רגשות לסדר הגעתן של התהליכים, זהו החיסרון של המдинיות הזה. ניתן לראות בדוגמה שכאשר התהליכים ארוכים מגיעים לפני התהליכים קצרים אז זה פוגע בזמן הממוצע של זמן המתנה.

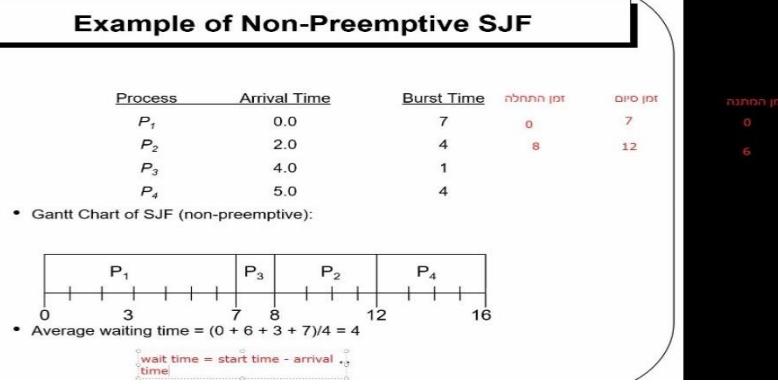
#### :SJF Shortest-Job-First Scheduling

אם נדע כמה זמן נשאר לכל תהליך לרווח אז נוכל להקצות את הזמן בצורה יعلاה יותר, לתת עדיפות לתהליכיים קצרים יותר לרווח, אך אין באמת דרך לדעת, אבל אפשר לנחש לפי סטטיסטיות עבר.

שנム שני סכומות:

- SJF (Non-preemptive) – בהנחה שה- scheduler יודע כמה זמן הולך לקחת כל תהליך, הוא ייתן העדפה לתהליך עם היכי קצר, לא עוצרים תהליך באמצעות ריצה.

## דוגמא:



שלבי

התהלייר:

- התהלייר הראשון שהגיע ה- Arrival Time מתחילה את התהלייר עד לסיומו.
- בינותיים תהלייר 2,3,4 נכנסו לו- Arrival Time וממתינים לסיום התהלייר הראשון.
- ה – Scheduler בוחר מבין תהלייכים 2,3,4 את התהלייר עם הקצר Burst Time.
- תהלייר 3 מתחילה ומסתיים.
- כעת נשארו תהלייכים 2,4 שלשניהם אותו Burst Time אחד מהם יבחר באופן שרירותי, ירוץ ויסתים ולאחריו התהלייר האחרון.

○ **Preemptive (SRTF)** – בדיקת SJF מלבד כך שהוא מאפשר

קיטועת התהלייר באמצע במקרה כדי לתת עדיפות לתהלייר קצר יותר. אם המשימה החדשה יותר קצרה מהמשימה הנוכחית אז נבחר לבצע את התהלייר החדש, התהלייר קצר ביותר אף פעם לא ימתין.

לאחר סיום התהלייר הקצר ביותר נחזיר לתהלייר שקטענו או שנתחיל בתהלייר חדש קצר יותר מהטהלייר שנקטע וחוזר חלילה.

### Example of Preemptive SJF (SRTF)

Process	Arrival Time	Burst Time	time 4:
$P_1$	0.0	7	$P_1$ : remaining time=5
$P_2$	2.0	4	$P_2$ : remaining time=2
$P_3$	4.0	1	$P_3$ : remaining time=1
$P_4$	5.0	4	$P_4$ : hasn't arrived yet

• Gantt Chart of SRTF (preemptive SJF):

wait times:  
 P1: between 2 to 11.  
 P2: between 4 and 5  
 P3: no wait  
 P4: between 5 and 7

• Average waiting time =  $(9 + 1 + 0 + 2)/4 = 3$

**SRTF הוא אופטימלי**, כלומר יתן את זמן המתנה הממוצע הכי נמוך עבור רשותה של תהליכיים.

### Determining Length of Next CPU Burst ♦

- ניתן רק לשער או להעריך את האורך של ה-CPU burst.
- ניתן לבצע Averaging Exponential כדי לשער את הזמן זהה, לכל איבר הבא ניתן משקל יותר נמוך מהקדום.

#### Determining Length of Next CPU Burst

#### Examples of Exponential Averaging

- Can only estimate the length.
- Can be done by using the length of previous CPU bursts, using exponential averaging.
  - 1.  $t_n$  = actual length of  $n^{th}$  CPU burst
  - 2.  $\tau_{n+1}$  = predicted value for the next CPU burst
  - 3.  $\alpha, 0 \leq \alpha \leq 1$
  - 4. Define :
$$\tau_{n+1} = \alpha t_n + (1 - \alpha) \tau_n$$
- $\alpha = 0$ 
  - $\tau_{n+1} = \tau_n$
  - Recent history does not count.
- $\alpha = 1$ 
  - $\tau_{n+1} = t_n$
  - Only the actual last CPU burst counts.
- If we expand the formula, we get:
 
$$\begin{aligned} \tau_{n+1} &= \alpha t_n + (1 - \alpha) \alpha t_{n-1} + \dots \\ &\quad + (1 - \alpha)^2 \alpha t_{n-2} + \dots \\ &\quad + (1 - \alpha)^{n+1} t_0 \end{aligned}$$
- Since both  $\alpha$  and  $(1 - \alpha)$  are less than or equal to 1, each successive term has less weight than its predecessor.

### Priority Scheduler ♦

לכל תהליך נקבע איזשהו עדיפות, ניתן לקבוע עדיפויות בכל מיני שיטות, בהתאם למה שמתכוון המערכת רוצה.  
 למשל FJS הוא סוג של תור עדיפויות כאשר העדיפויות נקבעות בהתאם ל- CPU Burst הקצר ביותר של תהליך מסוים.

- אם כל הזמן יכנסו תהליכים קצרים שモוגדרים **High Priority** – איזה תהליכיים שモוגדרים **Low Priority** לעולם לא יקבלו זמן לפעול, כמובן מדובר על תיזמן בתור עדיפות.
- הפתרון לבעה של Starvation הוא שיכל שתהליך מקבל יותר זמן פעולה ניתן לו עונש בכך שהוא priority שלו ירד.

### :Round Robin (RR) Scheduling ♦

שיטת תיזמן פשוטה ויעילה שאומרת שכל תהליך שנכנס מקבל איזשהו ייחידת זמן אשר מסומנת באות  $q$  ונבחרת באופן שרירותי, ככל שמקטינימ את  $q$  אז יהיו יותר Context Switching, אחרי שהזמן הזה עבר ישנה פסיקה, התהליך שהסתיים עובר לסופ התור וההתהליך הבא רץ עד הפסקה הבאה וכן האלה בתור בעגلي, ללא עדיפות בתהליך מסוים.

- בשיטה זו אין Starvation וזה אחת היתרונות שלה. פועל במערכות מודרניות.
- בדרך כלל זמן המתנה הממוצע בשיטה זו יותר גבוה מזו של SJF אך הרסpondency שלה תהיה יותר טובה בגלל שאין מצב של Starvation.

**Example: RR with Time Quantum = 20**

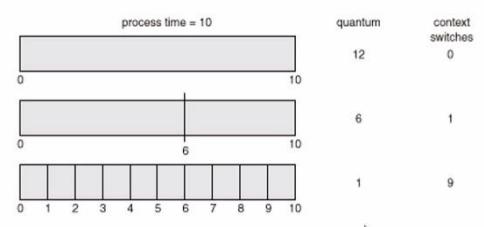
Process	Burst Time
$P_1$	53
$P_2$	17
$P_3$	68
$P_4$	24

- The Gantt Chart is:

$P_1$	$P_2$	$P_3$	$P_4$	$P_1$	$P_3$	$P_4$	$P_1$	$P_3$	$P_3$	
0	20	37	57	77	97	117	121	134	154	162

- Typically, higher average turnaround than SJF, but better response.

**How a Smaller Time Quantum Increases Context Switches**



## הרצאה 5

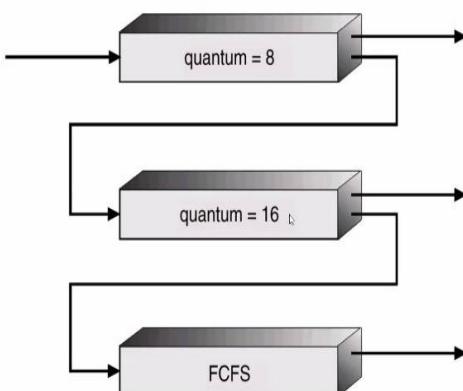
### ❖ תור רב שכבותי – Multilevel Queue

- ו ניתן לחלק את ה- Ready Queue לכל מיני מחלקות של תהליכיים, כאשר לכל תור יש את אלגוריתם ה- Scheduling Shallow (RR או FCFS או SJF או SRTF).
- בשיטת זו נדרש להחליט איך לחלק את הזמן בין התורים, אפשר לבחור בשיטת Fixed Priority כלומר לתת עדיפות לתהליכיים ב- Foreground או ל手続きים ב- Background אך יש סכנה לו - Starvation.
- לסיכום – ניתן לנתח תורים וכל אחד ניתן לבחור באיזה אלגוריתם יפעל.

### ❖ Starvation – דרך לפתרו את בעיית ה - Multilevel Feedback Queue

❖ כאשר יש  $X$  תורים אזי ניתנת האפשרות שתתהליך ינדוד בין התורים, למשל ניתן לראות שיש 3 תורים, השניים העליונים עובדים בשיטת RR והאחרון הוא תור שעובד בשיטת FCFS.

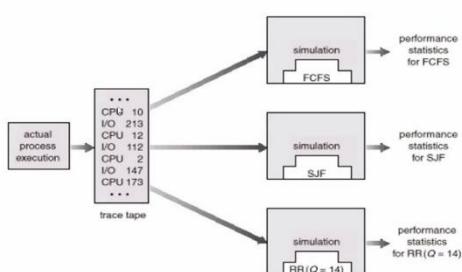
תתהליך חדש מגע ונכנס לתור הראשון, אם הוא לא הסתיים בזמן שהוקצב לו (8 מילישניות) אז הוא יקטע אחרי ה- Interrupt ויעבור לתור הבא שבו יש זמן של 16



밀ישניות לרוץ, אם הוא עדין לא סיים אז הוא יעבר לתור הבא של ה- FCFS שבו הוא מתinan עד שני התורים העליונים ריקים, כשהזה קורה הוא מקבל זמן בתור השלישי שעובד בשיטת ה- FCFS, כלומר קיבל זמן ריצה עד שהתתהליך יסתום. משתמשים זהה בשביל לנסוטה לסיים תהליכיים קצרים כמה שיותר מהר, ובשביל לפתרו את בעיית ה – Starvation, בעיקרו זה ניסiou ליעול.

### ❖ Algorithm Evaluation

#### Evaluation of CPU Schedulers by Simulation



אם אנו רוצים להשוות בין כמה אלגוריתמים (SJF, FCFS, RR...) אזי ניתן ללקחת רשימת תהליכיים (Workload) ולבצע סימולציות על כל אחד מהאלגוריתמים ולהחליט מי יותר קצר (מבחןת ה- Average waiting time) לרשות התהליכיים.

נכתב על ידי אוחד אמסלם, תשפ"ב.

## **:Linux Scheduling ♦**

לינוקס עובד בשיטת- **Multilevel Queue** בשלושה רמות -

- .FIFO Real-Time
- .RR Real-Time

○ Non-Real-Time scheduling – תור עבר על התהליכים שאינם בזמן אמיתי.

### **נרחיב על Non-Real-Time scheduling :**

- התהליכים עוברים בין התורים רק בפקודה מפורשת.
- השיטה בה אוחז מזמן את התהליכים בתור זהה הוא בעזרת חלוקת הזמן ל- **Epochs** כלומר פרקי זמן (תקופות), בתקופה הראשונה לכל תהליך יש איזושהי הקצאה לכמה זמן מותר לו לרווח, כלומר כמה Quantum הוא מקבל, הערך ההתחלתי של ה- Quantum מבוסס על ה- Priority שלו, ה- priority של תהליך ב- Auvon מבוסס על ה- Nice Value שלו, בכל תקופה ישנה הקצאה לכל תהליך של כמה זמן מותר לו לרווח.
- **Nice Value** – סוג של עדיפות, כלומר מספר בטוחה [20,19-] שmbetta מה ה- Priority שלו כאשר מספרים נמוכים הם יותר טובים וברירת המחדל היא 0.

זה נקרא Nice כי יש בלינוקס Call System Shnkarא Nice System Call Shnkarא Nice שפעילה אותו, בנוסף רוב התהליכים הם בערך 0 (ברירת מחדל) ואפשר להתחשב בתהליכים אחרים בצורה צו שנייתן לתהליך מסוים עדיפות נמוכה יותר, למשל 5, וזה מתחשב בתהליכים האחרים.

- **התקופה נגמרת כאשר אין יותר תהליכים ב- Ready Queue** שנשאר להם הקצאת זמן, כלומר או כאשר יש תהליכים שישימנו לרווח את הקצאת הזמן שלהם או שיש תהליכים שמחכים ל- 0/0 ובעצם לא דורשים זמן ריצה עד שהם מסיימים את בקשת ה- 0/0.

ברגע שהגענו למצב הזה ניתן לומר שתקופה הסתיימה ונפתחת תקופה חדשה.

- בתקופה החדשה כל התהליכים מקבלים הקצאה חדשה שמחושבת כך: הבסיס הוא לפי ה- Nice Value אבל יש בונוס לתהליכים שלא ניצלו את מלאה הרקצאה שלהם בתקופה הקודמת, הם מקבלים חצי מהזמן שלא נצל בתקופה הקודמת לשימוש בתקופה הנוכחית, זה בעצם מפיצה את התהליכים שלא ניצלו את מלאה זמןם בסביבה הקודם.

## **:Windows Scheduling ♦**

וינדוס עובד בשיטת ה- **Multilevel Queue** בשתי רמות:

- .Real-Time
- .Non-Real-Time

### **נribbon על :Non-Real-Time scheduling**

- התחלים עוברים בין התורים רק בפקודה מפורשת.
- בוינדוס ה- Priority נע בין 1-16 כאשר ערך קטן יותר מייצג יתר טוב.
- וינדוס מחלק בונוסים שנקראים Thread's Type, כיצד הבונוסים מוחולקים?
  1. תחלים שקשורים ל- Window Focus (החלון שכרגע נמצא בפקוס והמשתמש משתמש בו) מקבלים פי 3 יותר זמן ממנו שהם היו אמורים לקבל אם לא היו בפקוס.
  2. תחלים שהמтиינו ל- 0/0 מקבלים בונוס שהוא ביחס הפוך **למהירות של התקן ה- 0/0**, כלומר תחלים שהמтиין ל- 0/0 מהתקן מהיר יקבל בונוס קטן אבל תחלים שהמтиין ל- 0/0 מהתקן איטי (מקלדת, עכבר וכו') יקבל בונוס ענק.
  3. בכל Quantum הבונוס קטן עד שהוא נעלם.

הרצאה 6

## Process Synchronization

נניח שיש לנו שני תהליכיים שרצים במקביל ומשתפים זיכרון, והיה איזושהי פסיקה ממש לפני פועלות השמירה של החסורה, התהילה השני יבצע את הפעולה שלו אך הוא לא מכיר את החיסור כי התהילה הראשון לא הספיק לזכור את הפעולה אז במקומם להחסיר מהמספר המעודכן הוא מחסיר מהמספר הקודם שהוא טען בזיכרון, ניתן לחשב על משיכת סוף מכטבומט משני כרטיסי אשראי של חשבון אשראי.

לכו יש לנו אוצר בוטנאי של חפלייכים

- Suppose the value 1000 is stored in ADDRESS1 in main memory

- Suppose we try to run these processes in parallel:

<u>Process A</u>	<u>Process B</u>
load a, (ADDRESS1)	load a, (ADDRESS1)
suspend └ a-=100	a-=100
store a, (ADDRESS1)	store a, (ADDRESS1)

- If we start with process A and we suspend process A before the store command, our result will be 900 instead of 800.

## **משמעות:**

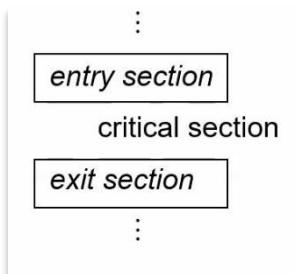
- יכול לגרום לחוסר עקביות בנתונים.
  - כדי לשמר על עקביות בנתונים נctrיך להבטיח את ההוצאה לפעול של תהליכי שמשתפים זיכרון, בסדר הנכון שלהם.
  - ניתן לטפל בבעיית ה- **Producer Consumer Processes** שדיברנו עליו בהרצאה 3, מבחינת המילוי של הבאפר באופן מלא במקום למלא את הבאפר ב- 1-a מהמקום.

ניתן לעשות זאת באמצעות הוספת 1 – counter + 1 – ב- **Consumer** ו- 1 – ב- **Producer**, אך לצורך הבטיח שכל פעולה מתבצעת בצורה אוטומטית כדי להימנע מהבעיה שהראננו לעללה, קלומר חוסר התאימות מבחינת תמונה הזיכרון המשותפת.

- **פעולה אוטומית** – פעולה שאנו יכולים להבטיח שהיא תבוצע בשלוםותה ללא הפרעה.
- **Race Condition** – זהו השם של הבעיה שהגדכנו למטה.

### **:The Critical-Section Problem**

- תהליכיים שמתחרים על הזמן המשותף.
- לכל קטע קוד יש אזור בקוד שנקרא **Critical Section** שבו יש גישה לדיזכרון משותף.
- נרצה שכאשר קטע הקוד נמצא באזורי זהה אז אף תהליך אחר לא יוכל להפריע לתהליך עד שהוא מסיים את הפעולה ב- **Critical Section**.



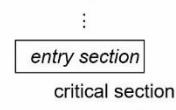
### **▪ פתרון לבנייה ה- Critical-Section**

1. **Mutual Exclusion** – כאשר תהליך רץ ונמצא ב- **Critical Section** שלו אף תהליך אחר לא יפריע לו.
2. **Progress** – אם אף תהליך לא רץ כרגע ב- **Critical Section** שלו ויש תהליכיים שרצוים להיכנס ל- **Critical Section** שלהם אז התהליך הבא שרצויה להיכנס לחלק הזה בקוד שלו לא יכחלה ללא סוף.
3. **Bounded Waiting** – אם יש לנו כמה תהליכיים שרצוים להיכנס ל- **Critical Section** שלהם אז נרצה שכל אחד מהם יוכל להיכנס לקטע הקוד הזה שלהם, שלא תהיה הרעבה של תהליך שרצויה להיכנס להיכנס לקטע הזה עצמו.

## הרצאה 7

### Initial Attempts to Solve Problem

- Only 2 processes,  $P_0$  and  $P_1$ .
- In  $P_0 \rightarrow i=0$  and  $j=1$ . In  $P_1 \rightarrow i=1$  and  $j=0$ .
- General structure of process  $P_i$  (other process  $P_j$ )



### Algorithm 1

- Shared variables:
    - var  $turn: (0..1)$ ;  
initially  $turn = 0$
    - $turn - i \Rightarrow P_i$  can enter its critical section
  - Process  $P_i$
- entry section → while  $turn \neq i$  do no-op;  
critical section  
exit section →  $turn := j$ ;
- Satisfies mutual exclusion and bound waiting, but not progress.

### Algorithms to Solve Critical Section Problem

1. ישנו שני תהליכי, כאשר אחד מהם

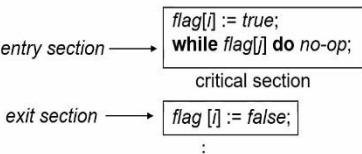
שווה 0 אז התהליך השני ממתין,

וכאשר התהליך הראשון מסיים את הפעולה שלו הוא משנה את הערך שלו  
ל- 1 וההתהליך השני מקבל רשות  
להיכנס ל- Critical Section .

הפתרון הוא פתרון נכון, קלומר אין מצב  
שני תהליכי יתכנסו ל- Critical Section ואין מצב של הרעבה, אבל  
הפתרון זהה לא מקיים Progress, לכן זהו פתרון לא טוב.

### Algorithm 2

- Shared variables
  - var  $flag: array [0..1] of boolean$ ;  
initially  $flag [0] = flag [1] = false$ .
  - $flag [i] = true \Rightarrow P_i$  ready to enter its critical section
- Process  $P_i$



- Satisfies mutual exclusion and bounded waiting, but not progress.

2. אם רק תהליך אחד רוצה להיכנס  
לקטע הקרייטי אז לא יהיה בעיות,  
במקרה שיש שני תהליכי ומעלה  
שרצים אז הדגל הראשון שווה 1,  
אם הדגל השני לא מורם, קלומר  
שווה 0 אז הוא יכול להיכנס לקטע  
הקרייטי, בסוף הריצה בקטע הקרייטי  
התהליך יוריד את הדגל ואז  
התהליך שרצה להיכנס יוכל להיכנס  
לקטע הקרייטי שלו.

הפתרון מקיים Mutual Exclusion ומקיים Bounded Waiting אבל לא  
מקיים Progress.

אם שניים במקורה מצליחים להרים דגל במקביל אז שנייהם יחכו לתמי'ץ.  
**לכן זהו פתרון לא טוב.** (Deadlock)

3. שילוב של אלגוריתמים 1 ו- 2, אם תהליך רוצה להיכנס אז הוא

### Algorithm 3

- Combined shared variables of algorithms 1 and 2.
- Process  $P_i$

```

entry section →   flag [i] := true;
                   turn := j;
                   while (flag [j] and turn = j) do no-op;
                           :
                           critical section
exit section →    flag [i] := false;
                   :

```

- Meets all three requirements; solves the critical-section problem for two processes.

mdl'ik את הדגל שלו והוא  
 ממתיין עד שהדגל של התהליך  
 השני הודלק וגם התור שלו על  
 0.

**הפתרון מקיים את כל שלושת**  
**התנאים,** Bounded, Mutual Exclusion  
 ו**וגם Progress עבור שני**  
**תהליכיים בלבד.**  
**זהו פתרון טוב.**

4. לפני כניסה לקטע הקרייטי התהליך מקבל מספר, התהליך עם  
 המספר הקטן ביותר נכנס לקטע הקרייטי.  
 אם המספרים שהם מקבלים זהים אז מי שקובע זה ה-ID  
 היוטר קטן.  
**זהו פתרון שעבוד עבור כל תהליכיים. לכן זהו פתרון טוב מאוד.**

### Bakery Algorithm

Critical section for n processes

- Before entering its critical section, process receives a number. Holder of the smallest number enters the critical section.
- If processes  $P_i$  and  $P_j$  receive the same number, if  $i < j$ , then  $P_i$  is served first; else  $P_j$  is served first.
- The numbering scheme always generates numbers in non-decreasing order of enumeration; i.e., 1,2,3,3,3,3,4,5...

### Bakery Algorithm (Cont.)

- Notation → lexicographical order (ticket #, process id #)  
 $(a,b) \prec (c,d)$  if  $a < c$  or if  $a = c$  and  $b < d$   
מספר תחילה, מספר שני
- Shared data
 

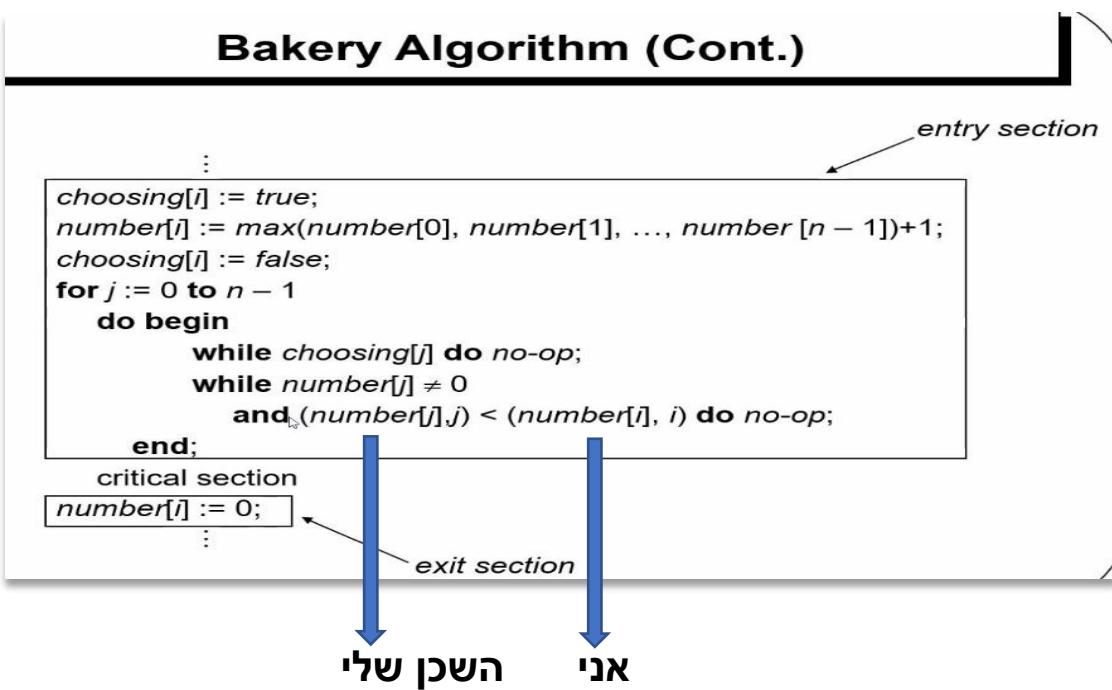
```

var choosing: array [0..n - 1] of boolean;
number: array [0..n - 1] of integer;
```

Data structures are initialized to *false* and 0 respectively.

בתחילת האלגוריתם המערך `number` מלא באפסים והתהליך הראשון מקבל את המספר 1, כאשר תהליך נוסף ירצה להיכנס לקטע הקרייטי שלו והוא יקבל את המספר 2 וימתין עד שהתהליך הראשון יסיים לרווח את הקטע הקרייטי שלו, כלומר עד שהוא יגדיר את עצמו ל-0 ועד שהמספר מאפייה של השכן שלו גדול מהמספר מאפייה שלו.

## **Bakery Algorithm (Cont.)**



## -Synchronization Hardware •

אפשרויות יותר מהירות לסנכרון הן לבצע סנכרון באופן חומרתי בצורת *Test and*

## Mutual Exclusion with Test-and-Set

- Shared data: `var lock: boolean (initially false)`
  - Process  $P_i$

```

sequenceDiagram
    participant ES as entry section
    participant CS as critical section
    participant LS as lock section
    ES->>CS: while Test-and-Set (lock) do no-op;
    activate CS
    CS->>LS: lock := false;
    deactivate CS

```

The diagram illustrates a sequence of operations for managing a critical section:

- entry section** sends a message to the **critical section**: **while Test-and-Set (lock) do no-op;**
- The **critical section** enters a loop where it repeatedly attempts to **Test-and-Set (lock)**. If successful, it proceeds to the next step; if not, it loops back to the test.
- critical section** sends a message to the **lock section**: **lock := false;**
- The **critical section** exits the loop and continues with the rest of its code.

- A possibility of starvation.
  - There is a problem of busy waiting.

שנתפקיד שלה זה Set  
לקראת הערך הנוכחי  
מיצושהו מילה בזיכרון  
ולכתוב לשם True, זה  
מפרקנו בזרוג אנטומום

- **ל- Bounded Waiting** מטבח בצורה אוטומית, כלומר במקה אחת.

כתב על ידי אוד אמלט. משפט ב-

## • דרך נוספת, שימוש בפעולת האוטומית Swap:

החלפה בין התהליכים מתבצעת במקה אחת, כל עוד הוא עושה החלפה של True עם False הム Ichco ב- While עד שההתהילר (lock) מסיים את הפעולה שלו בקטע הקרייטי ויגדר את עצמו כ- False.

### Another Synchronization Hardware

- Atomically swap two variables.

```
procedure swap (var a, b: boolean);
begin
    var temp: boolean;
    temp := a;
    a := b;
    b := temp;
end;
```

### Mutual Exclusion with Swap

- Shared data (initialized to false):

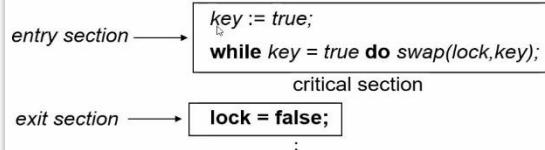
```
var lock: boolean;
```

- Unshared data

```
var key: boolean;
```

- Process  $P_i$

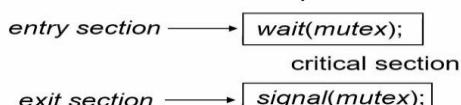
:



## • אפשרות נוספת לפטור את בעיית הקטע הקרייטי הוא להשתמש במערכת

### Semaphore

- Synchronization tool that implemented in many operation systems and does not require busy waiting.
- Shared variables
  - **var mutex : semaphore**
  - initially **mutex = 1**
- Process  $P_i$



### Semaphore Implementation

- Define a semaphore as a record

```
type semaphore = record
    value: integer;
    L: queue of process;
end;
```

- Assume two simple operations:
  - block suspends the process that invokes it.
  - wakeup( $P$ ) resumes the execution of a blocked process  $P$ .
- can only be accessed via two indivisible atomic operations:
  - wait
  - signal

### הפעלה (Semaphore)

עושה זאת באמצעות System Calls.

המתנה נעשית על ידי זה שמערכת הפעלה שמה תהילר במצב של Waiting עד שתהילר אחר יבצע Signal.

### Implementation (Cont.)

- Semaphore operations now defined as

```
wait(S): S.value := S.value - 1;
```

```
if S.value < 0
```

```
then begin
```

```
add this process to S.L;
```

```
block;
```

```
end;
```

```
signal(S): S.value := S.value + 1;
```

```
if S.value ≤ 0
```

הזהר נא ציך

```
then begin
```

```
remove a process  $P$  from S.L;
```

```
wakeup( $P$ );
```

```
end;
```

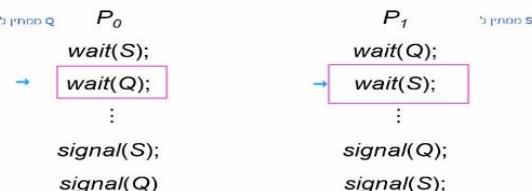
## • -Semaphore

בכמויות וסדר התהילכים  
שרצים.

### Deadlock and Starvation

S=1  
Q=1

- Deadlock – Two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes.
- Let S and Q be two semaphores initialized to 1



- Starvation – indefinite blocking. There is no way to know how much processes will get the CPU before a starved process.

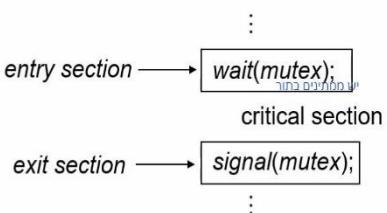
סמאפור הוא איזשהו מספר  
תו, כל עוד הערך של  
הסמאפור הוא חיובי אז  
טהילכים יכולים לעשות Wait  
ואם הוא בערך שלילי אז הוא  
לא יכול לעשות Wait הוא יכול  
לעשות Signal.

כasher משתמשים בו- Semaphore צריך להיזהר כי ניתן להגיע ל- Deadlock .

## ◦ שימוש ב- Semaphore לפתרון בעיית הקטע הקרייטי:

### Semaphore

- Synchronization tool that implemented in many operation systems and does not require busy waiting.
- Shared variables
  - var mutex : semaphore
  - initially mutex = 1
- Process  $P_i$



### Implementation (Cont.)

- Semaphore operations now defined as

```

wait(S): S.value := S.value - 1;
if S.value < 0
then begin
      add this process to S.L;
      block;
end;
signal(S): S.value := S.value + 1;
if S.value ≤ 0
then begin
      remove a process P from S.L;
      wakeup(P);
end;
    
```

## ♦ Semaphores ב- Linux שצריך לזכור בנוגע ל- System Calls ♦

.Semaphores – מייצר סט Semget .1

.Semctl – מבצע פעולות, קביעת ערך וכו' .2

.Semop – עושה פעולות Wait ו- Signal .3

ו. שימוש ב- **Semaphore** ככלי סינכרונייזציה רגיל:

## Semaphore as General Synchronization Tool

I

- Execute  $B$  in  $P_j$  only after  $A$  executed in  $P_i$
- Use semaphore  $flag$  initialized to 0
- Code:

$P_i$	$P_j$
⋮	⋮
$A$	$wait(flag)$
$signal(flag)$	$B$

**wait:**  
אפס ומטה - המהינה  
אחד ומעליה - משוחרר פירמידת

ערך חיובי: מציין את מספר התאים הפנויים  
אפס: כל התאים בשימוש, וגם יש מנתנים  
ערך שלילי: כל התאים בשימוש, וגם יש מנתנים



▪ אם Semaphore יהיה על 0 ומגיע Wait לפני Signal אז תהייה המתנה.

## הרצאה 8

- **שניהם שני סוגים של Semaphore**
  - **Counting Semaphore**
  - **Binary Semaphore**

### Implementing S as a Binary Semaphore

- Data structures:

```
var S1: binary-semaphore;
    S2: binary-semaphore;
    C: integer;
```

- Initialization:

```
S1 = 1
S2 = 0
C = initial value of semaphore S
```

### Implementing S (Cont.)

- wait operation:

```
wait(S1);
C := C - 1;
if C < 0
then begin
    signal(S1);
    wait(S2);
end
else signal(S1);
```

- signal operation:

```
wait(S1);
C := C + 1;
if C ≤ 0 then signal(S2);
signal(S1);
```

### – Bounded-Buffer Problem (Producer Consumer) with Semaphores

ניתן להשתמש ב-Semaphore לפתרון בעיית-Producer Consumer שראינו בהרצאות קודמות בצורה הבאה:

### Bounded-Buffer Problem

נשתמש ב-3 Semaphores: **full**, **empty**, **mutex**

1. נתחל את full ל-0 כאשר

**full** מייצג את מספר התאים המלאים.

2. **empty** מייצג את מספר התאים הריקים והוא מאותחל ל- n תאימים.

- Shared data

```
type item = ...
var buffer = ...
    full, empty, mutex: semaphore;
    nextp, nextc: item;
    full := 0; empty := n; mutex := 1;
```

זה Semaphore שותפים לרגע רק כדי לעדכן את הקונטרארים. 3

## - Producer Process

1. מייצרים item.
2. ממתינים על ה- Semaphore שנקרא empty, ה- Wait הראשו משתחרר מיד.
3. לאחר מכן אנו תופסים את mutex.
4. מאחסנים את הפריט שיצרנו במערך.
5. משחררים את ה- mutex.
6. עושים Signal ל- Semaphore שנקרא full, כי אנחנו רוצים להגדיל ב-1 את מספר התאים שמסומנים כתפוסים במערך כי התהילה שוצרק את היחידות העבודה (התהילה שמרוק) הולך להמתין עד שהיא משה במערך, כלומר עד full יהיה גדול מ-0 ואז הוא ישתחרר.

## **Bounded-Buffer Problem (Cont.)**

- Producer process

```

repeat
    ...
    produce an item in nextp
    ...
    wait(empty);           המתינה עד שהיא מקום פנוי במערך
    wait(mutex);          הוגנה על משתנים כפוי
                           ...                                     בו
                           ...                                     מוכניס למערך
                           ...
                           signal(mutex);           מעדכן שיש איבר אחד יותר במערך
                           signal(full);          ...
until false;

```

## - Consumer Process

עושה את אותו התהילה שעשו ה- Producer רק בתמונה מראה.

1. המתנה על ה- Semaphore full.
2. הוגנה על משתנים כמו.
3. מסירים ערך מהמערך.
4. עושה Signal על הסמפור empty.
5. צורק את האיבר הבא.
6. כל עוד יש איברים במערך.

## Bounded-Buffer Problem (Cont.)

```
• Consumer process
repeat
    wait(full)
    wait(mutex);
    ...
    remove an item from buffer to nextc
    ...
    signal(mutex);
    signal(empty);
    ...
    consume the item in nextc
    ...
until false;
```

– נאמר שיש אוזשה משאב, למשל קובץ.

החוקים לגישה למשאב זהה הם:

- יש אפשרות לקרוא ממנו ויש אפשרות לכתוב אליו
- לא מפריע לנו שייהיו קוראים במקביל, כלומר יכולם 100 להיכנס ולקראות נTONIM במקביל.
- אם מישו כותב אחד נרצה שהוא רק כותב אחד ושלא יהיו קוראים.

ישנם שני Semaphores, אחד שנקרא mutex ואחד שנקרא wrt (כתיבה), ויש משתנה שנקרא readcount (מספר הקוראים).

## Readers-Writers Problem

### • Shared data

```
var mutex, wrt: semaphore (=1);
readcount : integer (=0);
```

### • Writer process

```
wait(wrt);
...
writing is performed
...
signal(wrt);
```

אם ה-Semaphore של הכתיבה (wrt) תפוס אז זה אומר שיש כותבים שכרגע רוצים לכתוב, וה-readcount סופר כמה קוראים יש.

אם יש תהליך שרוצה לכתוב הוא מנסה לתפוס את הסמאפור wrt אם הוא מצליח אז הוא יכול לגשת למשאב ומובטח לו שאף אחד אחר לא יגש, איך מובטח לו? כי אם יבוא תהליך אחר שinessה לכתוב הוא לא יוכל להיכנס כי הסמאפור מאוחTEL ל-1.

## Readers-Writers Problem (Cont.)

- Reader process

```
wait(mutex);
readcount := readcount +1;
if readcount = 1 then wait(wrt);
signal(mutex);
...
reading is performed
...
wait(mutex);
readcount := readcount - 1;
if readcount = 0 then signal(wrt);
signal(mutex);
```

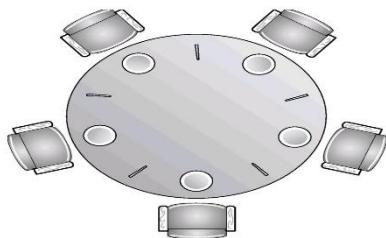
wrt רק הקורא הראשון תופס את wrt

- Possibility of starvation to the writer processes.

## Dining-Philosophers Problem

לأكل את הקערת אורך במרכז השולחן יכול להרים צ'ופסטיקו ולאכול (סך הכל 5

### Dining-Philosophers Problem



- Shared data

```
var chopstick: array [0..4] of semaphore;
(=1 initially)
```

מקלות כאשר סני צרי שמי מקלות כדי לאכול).

יש פה בעיית Deadlock כי אם כולם רעבים באותו רגע אז כל אחד ירים את הצ'ופסטיק שמיימנו (או משמאלו) אז אף אחד לא יוכל לאכול. הפתרון הכי טוב לבעה זה להרים רק אם יש שני צ'ופסטיקו פנויים, אחד מכל צד.

## Dining-Philosophers Problem (Cont.)

- Philosopher  $i$ :

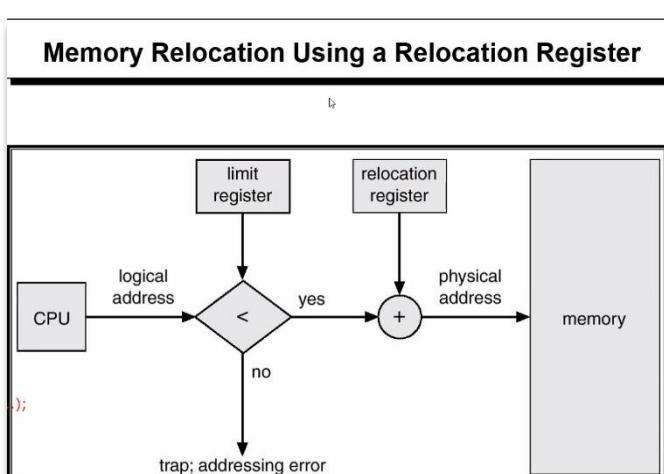
```
repeat
    wait(chopstick[i])
    wait(chopstick[i+1 mod 5])
    ...
    eat
    ...
    signal(chopstick[i]);
    signal(chopstick[i+1 mod 5]);
    ...
    think
    ...
until false;
```

- Possibility of deadlock.

## הרצאה 9

- תהליך חייב להגיע לדיסק חיצוני כדי לראז. **Memory Management** •
- חיבור כתובת לתהליך יכול **Binding of instruction and Data to Memory** •
  - לקרות ב 3 מקרים:
    - זמן קומpileציה. **Compile time** 1.
    - מיקום הזיכרון לא ידוע בזמן קומpileציה. **Load Time** 2.
    - בזמן ריצה התהליך יכול לעבור בין מקום אחד בזיכרון 3. למקום אחר. **Execution Time**
- ישנה הפרדה בין זיכרון לוגי לזיכרון פיזי.
  - ויתן לחשב על הזיכרון הפיזי כמערך של בתים שאפר לגשת לכל בית ולשלוך אותו, הזיכרון הפיזי זה מה שהחומרה רואה.
  - מערך זיכרון וירטואלי (לוגי) זה מה שהתוכנית משתמש רואה, תוכנית משתמש לא יודעת מה זה כתובות פיזית, למשל כאשר מתבצע הקצאת זיכרון ב – c אז אנו ניגשים לכתובות לוגיות.
- Memory Management Unit (MMU)** – ממפות בין כתובות וירטואליות לכתובות פיזיות בזמן ריצה, אם דף הזיכרון הנדרש לא נמצא בעט בזיכרון הפיזי, היחידה מייצרת פסיקה שמסורתית נקרה page fault ומערכת הפעלה מטפלת בפסיקה על ידי טעינת הדף מהזיכרון המשני, בדרך כלל דיסק או SSD.

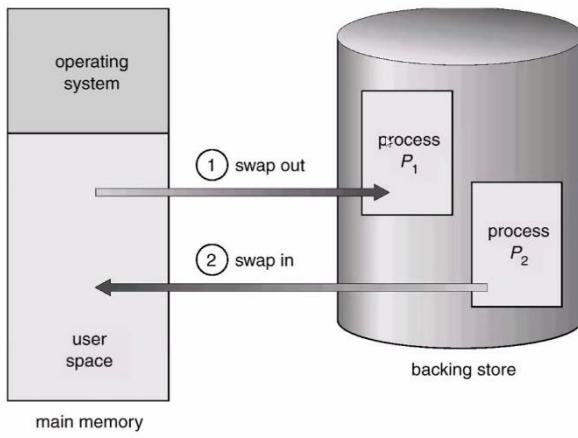
### – החומרה בודקת האם **Memory Relocation Using a Relocation Register** •



הכתובת הלוגית שמתחליה מ- 0  
תקינה, אם לא אז ישנו איזשהו Trap,  
אם כן אז אנחנו מוסיפים איזשהו ערך  
קבוע ואז מזידים את הכתובות  
הפיזיות.

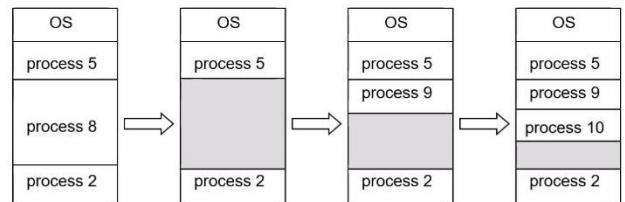
- **Swapping** – כאשר אנו רוצים להריץ הרבה תוכניות ביחד אך אין לנו מספק מקום בזיכרון הפיזי אז נעשה סוג של תרמיית, נחזיק חלק מתמונות תהליכיים בזיכרון הפיזי וחלק בדיסק, כאשר יש צורך להוריד עומס מהזיכרון הפיזי אז נבצע Swap ונעביר אותו לדיסק.
- כאשר נבצע הרבה החלפות אנו נבזבז זמן, חיסרונו נוסף הוא שהקצתת זיכרון לכל תהליך חייבת להיות רציפה. זו שיטה שכירום אינה בשימוש.

### Schematic View of Swapping



### Contiguous Allocation (Cont.)

- Multiple-partition allocation
  - *Hole* – block of available memory; holes of various size are scattered throughout memory.
  - When a process arrives, it is allocated memory from a hole large enough to accommodate it.
  - Operating system maintains information about:
    - a) allocated partitions
    - b) free partitions (hole)



- **Paging** – זה הפתרון העדכני של הבעיה מהעמדת הקודם, לכל תהליך יהיה מרחב זיכרון לוגי וזכרון פיזי יוקצה לזכרון פיזי כאשר יש צורך.
- מחלקים את הזיכרון הפיזי לבלוקים קבועים שנקראים Frames המכילים דפים ובדרכן כל גודלו של דף מסוים הוא 4kb אז אשר הגודל של הפריטים הוא בזכרון הפיזי והזיכרון הלוגי.
- אם יש תוכנית שתוטסת 12kb יש לנו 4 דפים אז לכל דף יהיה 3kb, גודלו של דף תמיד יהיה בחזקה של 2 כדי שלא נבזבז מקום.
- ישנה טבלה שמרכזת את כל הכתובות של הדפים.

#### – Fragmentation ○

- **External Fragmentation** – הבלוקים של הזיכרון הפיזי מפוזרים בדיסק.
- **Internal Fragmentation** – חיברים להקצות בכפולות שלמות של גודל דף.

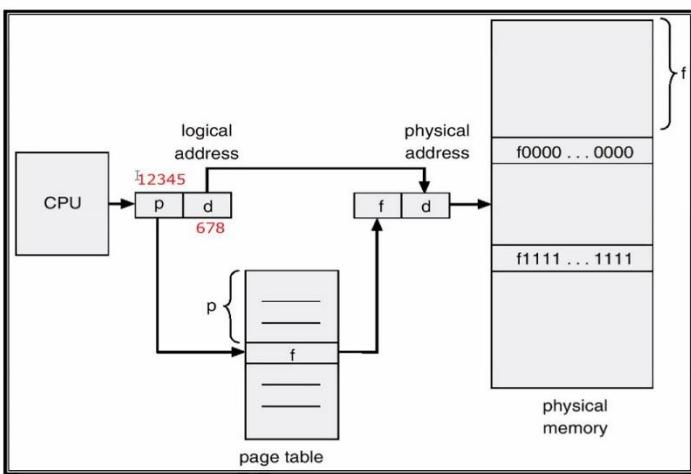
- ישנה בעיית O/I מכיוון שרכיב O/I רואה רק זיכרון פיזי ואינו מודע לזיכרון הלוגי, ניתן לפתור בעיה זו בעזרת נעלית הדפים בזיכרון בזמן שה-O/I מתבצע.

## Fragmentation

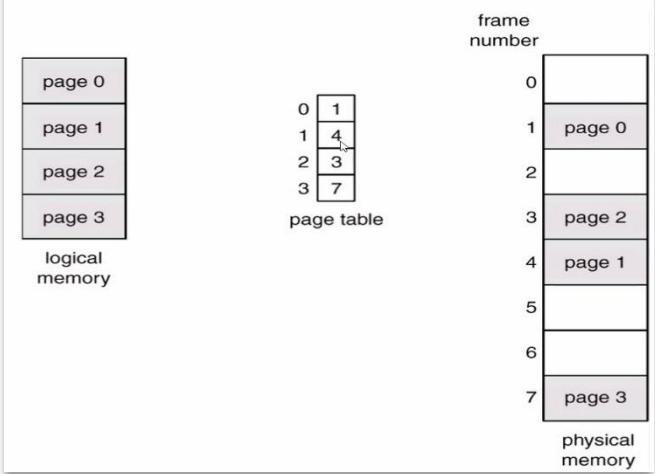
- External Fragmentation – The total memory space exists to satisfy a request, but it is not contiguous.
- Internal Fragmentation – The allocated memory may be slightly larger than the requested memory.
- Reduce external fragmentation by compaction
  - Shuffle memory contents to place all free memory together in one large block.
  - Compaction is possible *only* if relocation is dynamic, and is done at execution time.
  - I/O problem
    - Latch job in memory while it is involved in I/O.
    - Do I/O only into OS buffers.

- כתובות אשר מיוצרת בעזרת ה-CPU ( כתובות לוגית ) מתחולק בצורה הבאה:
  - מס' דפים - Page Number(p) ◦
  - אופסט בתוך הדף, מוביל לכתובת הפיזית. – Page Offset(d) ◦
  - מס' הכניסות בטבלת הדפים יהיה שווה למספר הדפים.

### Address Translation Architecture

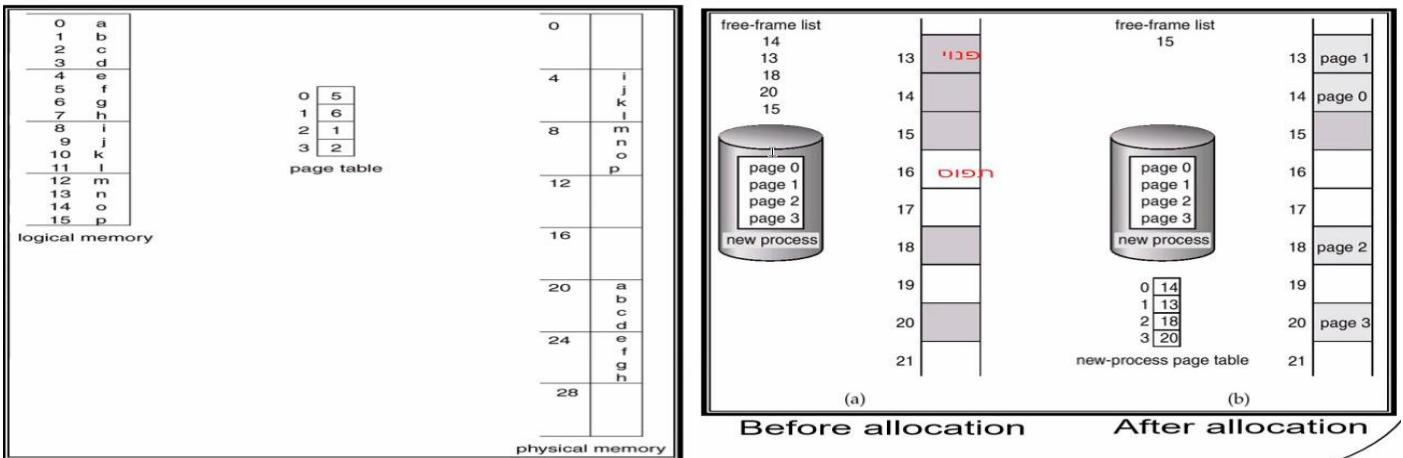


### Paging Example



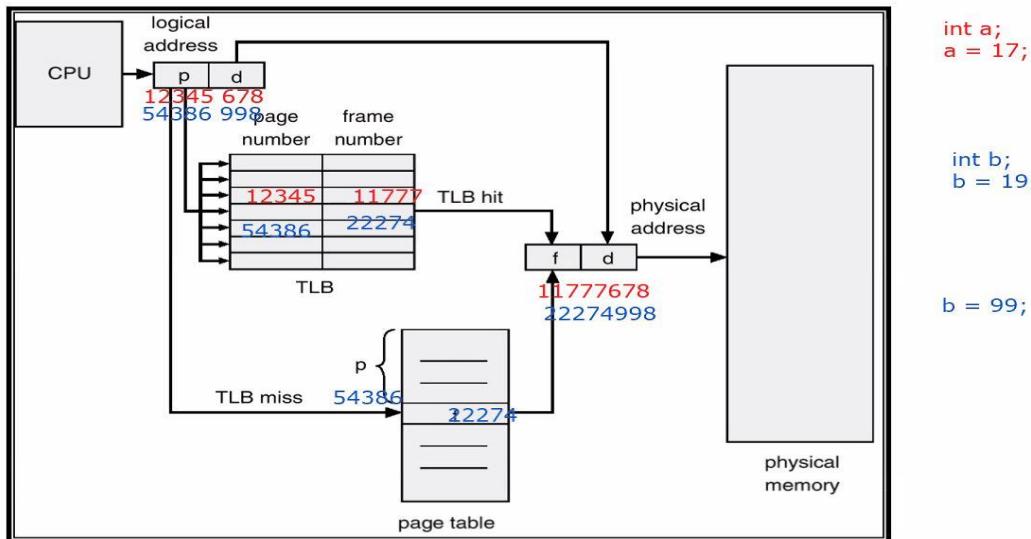
נכתב על ידי אוֹהָד אַמְסָלִם, תשפ"ב.

## Yet other Paging Examples



- טבלת הדפים עצמה נמצאת בזיכרון.
- טבלת הדפים היא פר תחלה.
- בכל רגע נתון יש לנו **PTBR** שהוא פוינטר לראש הכתובת של טבלת הדפים.
- כל גישה לזכרון תדרש שתי פניות לזכרון, אחד לזכרון הפיזי ואחד לטבלת הדפים, מה שמעט את הזמן הכלול לגישה לזכרון.
- כדי לפתור את הבעיה מהבולט הקודם, ניצור איזשהו Cache (שייתר מהיר מזכרון הראשי) שמאחסן חלק מהאינפורמציה שנמצא בטבלת הדפים, זה נקרא **TLB**.

## Paging Hardware With TLB



נכתב על ידי אוֹהָד אַמְסָלִם, תשפ"ב.

נרצה שיהיה לנו Hit ב- 90 אחוז מהזמן.  
כאשר עושים Context switch איז מאפסים את ה-TLB.

## Effective Access Time

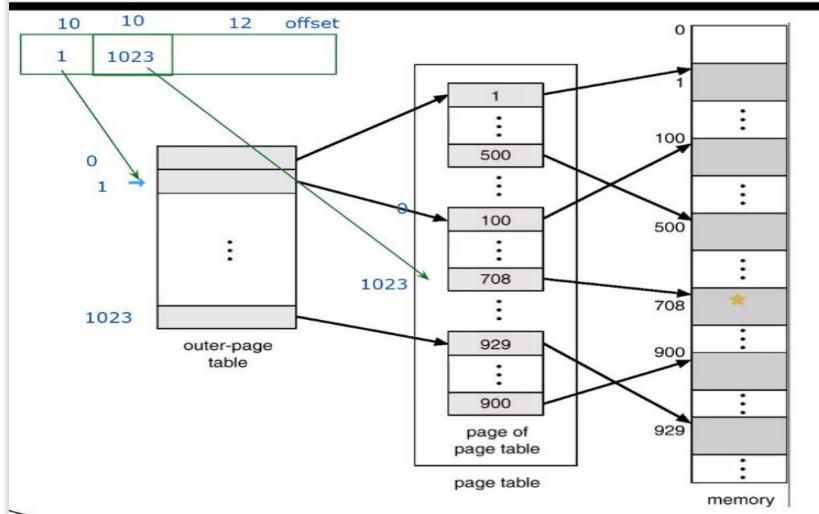
- Associative Lookup =  $\varepsilon$  microseconds
- Assume memory reference (average) time is 1 microsecond. (If page is in the disk, it will take much more time, If values are in cache, it will take less time so we take an average time)
- Hit ratio – percentage of times that a page number is found in the associative registers; ratio related to number of associative registers.
- Hit ratio =  $\alpha$
- Effective Access Time (EAT)

$$\begin{aligned} \text{EAT} &= (1 + \varepsilon) \alpha + (2 + \varepsilon)(1 - \alpha) \\ &= 2 + \varepsilon - \alpha \end{aligned}$$

### זמן הגישה האפקטיבי לזיכרון TLB

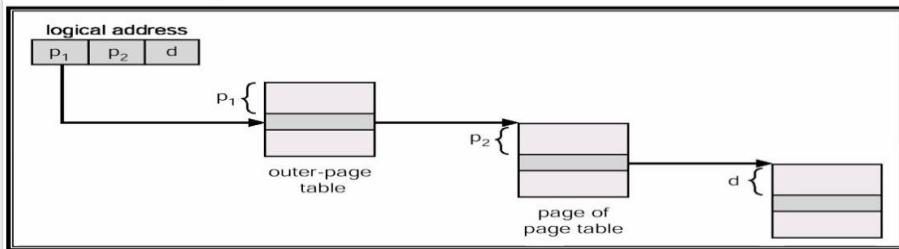
- טבלה דו שכבותית, נעשה בזה שימוש כדי לחסוך במקומות שתופסת הטלבת TBL.**
- אם חלק מהמרחב זיכרון לא מנוצל ניתן רק חלק מהטבלאות ולא להקנות זיכרון לטבלה שיש בה זאת, כך בעצם בה ידי ביטוי החיסכון בזיכרון. בסכמה שמצווגת פה הבלוק האחרון מיצג את הזיכרון הפיזי.

### Two-Level Page-Table Scheme



## Multilevel Paging and Performance

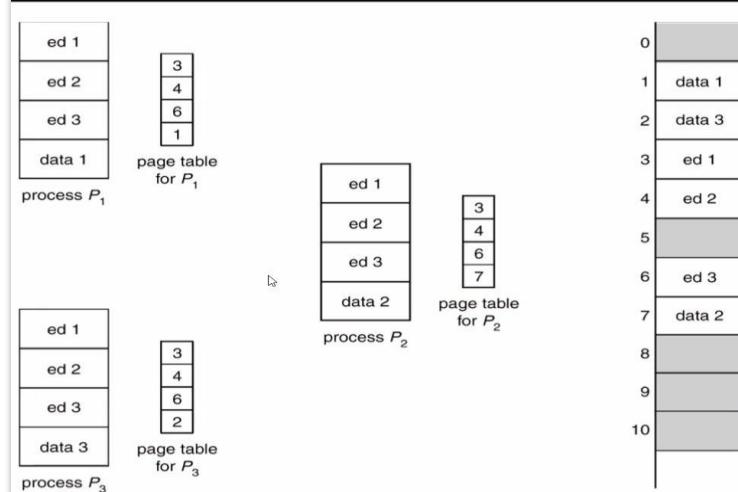
- Since each level is stored as a separate table in memory, converting a logical address to a physical one, may take more memory accesses.
- Even though time needed for one memory access is multiplied, caching permits performance to remain reasonable if Cache hit rate will be high.



## הרצאה 10

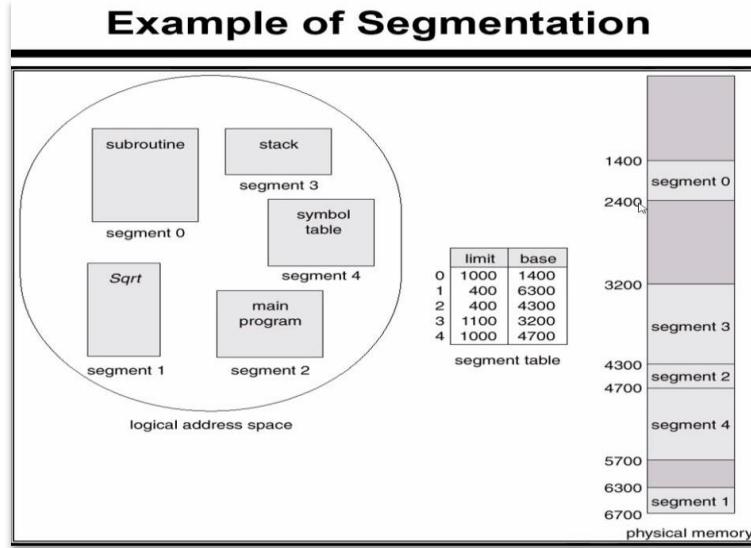
- **Shared Pages** - מאפשר גמישות וחיסכון בגישה לציכרונות. הדף הוא תמיד בגודל קבוע וזה אומר שכל דף יכול להתאים לכל פרויקט ולכל אם יש לו פרויקטים פנויים בזיכרון ניתן להשתמש בהם.

## Shared Pages Example



בדוגמה הנ"ל יש חיסכון בכך שלא טענים לזכרון תהליכיים שכבר רצים בזיכרון.

- מאפשר חלוקה של הזיכרון לגודלים מסוימים במטרה שתוכנית תוכל להיות מפוצלת בכמה חלקים בזכרון. זו שיטה פחות טובה מ- Paging כי יש אופציה שייצור חורים בזכרון.



### :Virtual Memory

ניתן לא לטעון בפועל דף לזכרון עד שתהיה גישה בפועל, כלומר במקום לטעון הכל לזכרון אך נחכה לרגע האחרון ורק ממש לפני שאני צריך אז אני אטען את מה שאינו צריך לזכרון, זהגורם לניצול זיכרון יותר טוב.

בגלל שיטה זו אנו לא צריכים את כל הזיכרון הדרוש כדי להריץ תוכנית מסוימת.

ניתן לישם Virtual Memory באמצעות:

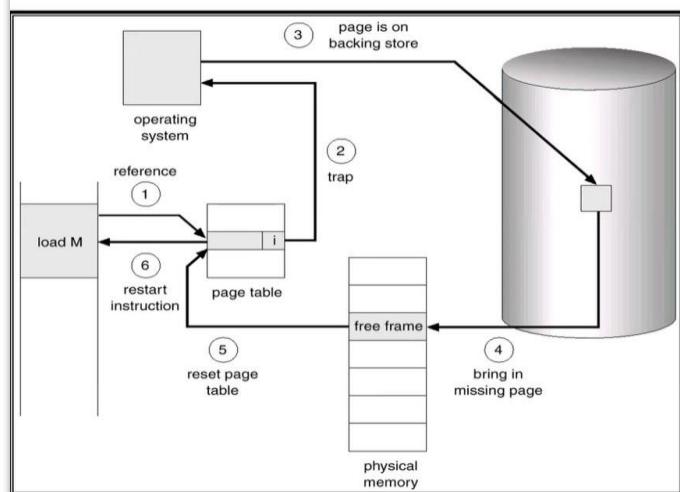
1. **Demand Paging** - הבאת דף לזכרון רק כאשר נדרש (מס נדחה הוא מס נחסך), חוסך בזכרון, התגובה יותר מהירה, מאפשר ליותר משתמשים לעבוד כאשר יש הגבלת בזכרון הפיזי.

כאשר יש גישה לאיזשהו דף החומרה תבודוק האם הדף הזה נמצא בזכרון, אם הוא נמצא אז תהיה גישה רגילה כמו שראינו בटבלת דפים.

אם הדף לא נמצא בזכרון אז החומרה תפנה למערכת הפעלה שתבודוק מה הסיפור, ישנו שני אפשרויות, האחת שהגישה זו לא חוקית בכלל, כלומר ניסיון גישה למקום בזכרון שלא הוקצה, השני הוא שהוא לא נמצא בזכרון אז יקרה מצב של **Page Fault**, לאחר מכן מערכת הפעלה תביא את הדף חזרה לזכרון ויהיה **ניסיון גסוף** לטעון את אותו הדף לזכרון פעם נוספת.

המימוש מתבצע בעזרת Bit **Valid-Invalid** (בית בקרה) בכל כניסה בטבלת דפים שיכל לקבל כਮון ערך 1 או 0 ומה שהוא מסמן זה האם הדף הזה נמצא בפועל בזיכרון כאשר 1 אומר שכן ו-0 אומר שלא.

## Steps in Handling a Page Fault



במצב בו ה- MMU מנסה לגשת לכניסה מסויימת בטבלת הדפים כדי לטעון קטע מסוים אז הדבר הראשון שהוא יעשה זה יבודק את הביט הזה, אם יש 1 אז הוא יטען את הקטע הרצוי, אם יש 0 אז החומרה (MMU) מייצר כולם מייצר פסיקה שגורמת לkapicha לקוד של מערכת הפעלה, ה- ISR (הfonkzia) במערכת הפעלה שמטפלת ב- (Page Fault) תבדוק מה הסיפור ותטפל בבעיה, לאחר שהיא הביאה את הקוד מהדיסק אז הביט בטבלת דפים ישנה ל-1 ויהי ניסין נוסף לטעון את אותו דף.

סבירות נוספת שיש בכל כניסה היא סיבית שנקראת **Dirty Bit**, מה שאומרת לנו הסיבית הזה אם היו כתיבות לדף הזה או לא.

למה אכפת לנו אם היה כתיבה? לפנות דף נקי מהזיכרון עולה הרבה פחות מאשר לפנות דף שהוא לו כתיבה וזה עוזר בבחירה קורבן (איזה חלק להעיף לדיסק כי אין לנו מקום).

כל שיש לנו יותר זיכרון פיזי אז יהיה לנו פחות Page faults.

## 2. Demand Segmentation

**Copy On Write** – מאפשר לתוכליי אב ובן לשתף מידע של טבלת הדפים שלהם. מערכת הפעלה מאפשרת לעשות זאת באמצעות שכפול של טבלת הדפים בין האב לבן, כאשר זה קורה אז נוסף בית נוסף ליד כל כניסה בטבלת הדפים שנקרו – **Read Only**.

אם ישנו ניסין כתיבה לדף שמסומן Read Only אז יהיה Page Fault, ככלומר יופעל Trap למערכת הפעלה, מערכת הפעלה תטפל בהז בכר שתקצה מקום חדש בזיכרון לאותו דף שהיה לו ניסין כתיבה. זה חוסך בזכרון כי ברוב המוחלט אנו לא צריכים לשכפל את כל הדפים, וחסכנו זמן.

**Memory-Mapped Files** – ניתן לבקש מערכת הפעלה למפות קובץ מסוים שנמצא בדיסק, לזכרון.  
זה מאפשר לנו לגשת לזכרון ולהשתמש בקובץ כאילו הוא משתנה בזיכרון.  
MMAP מאפשר לתהליכים לשתף מידע.

– **Page Replacement** – למצוא את המיקום בדיסק של הדף שאנו רוצים לטעון, כדי להביא אותו נצרך למצוא איפה הוא נמצא בדיסק, צריך למצוא מסגרת פנינה, אם יש מסגרת פנינה מצוין, אם אין מסגרת פנינה נצרך למצוא קורבן ולהעיף אותו אז נצרך אלגוריתם שמחליט מי הוא הקורבן, אחרי שעשינו Out Page אז אנחנו במצב שיש פריים פנוי, כעת אנו צריכים לטעון מהדיסק את הדף הרצוי לתוכר הפריים הפנוי, לעדכן את הטלאות דפים ולקפוץ שוב לתוכנית משתמש באותו מקום שהוא יצליח.  
ולהריץ אותו שוב בתקופה שהפעם הוא יצליח.

.**Page Faults** – נרצה אלגוריתם שגורם להכי פחות Page Replacement Algorithms

1. **FIFO** – **הדף הראשון יבחר הקרוב**, יותר זיכרון פיזי לא בהכרח מביאה לביצועים יותר טובים מבחינת Page Fault, האנומליה זו נקראת **Belady's Anomaly**.

### First-In-First-Out (FIFO) Algorithm

- Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
- 3 frames (3 pages can be in memory)
 

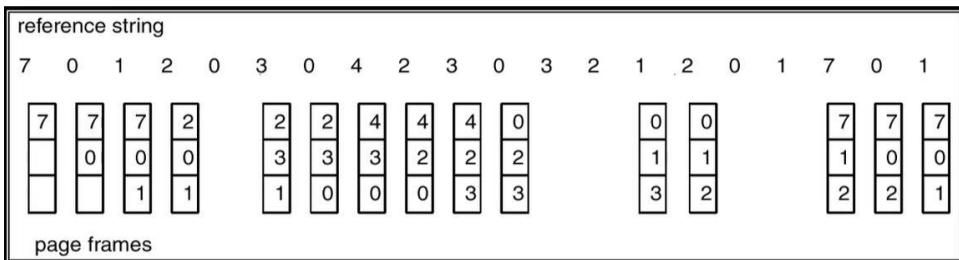
1	1	4	5
2	2	1	3
3	3	2	4

 9 page faults
- 4 frames
 

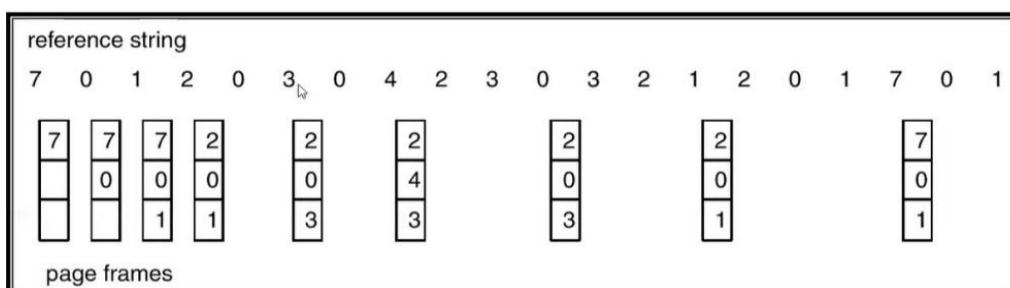
1	1	5	4
2	2	1	5
3	3	2	
4	4	3	

 10 page faults
- FIFO Replacement – Belady's Anomaly  
– more frames  $\Rightarrow$  less page faults

דוגמא נוספת לבחירת הקורבות כדי לפנות מקום לדפים אחרים:  
נשים לב שתמיד הדף הותיק מוסר.

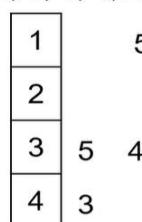


.2. קיימ אָלגוֹרִיתֵם אָוְפְּטִימָלִי אֲבָל רָק בָּמִידָה וְהִיא לֹא כָּדוֹר בְּדוּלָה שֶׁמְאָפָּשָׂר לֹא לְהַסְתַּכֵּל קְדִימָה מֵי הִיה הַכִּי פְּחוֹת בְּשִׁימוש אֵז נְבַחֵר בְּדָף הַזֶּה, אֲךָ אֵין לֹא בְּאַמְתָּא אֲפָּשָׂר לְעַשּׂוֹת אֶת זֶה. אָלגוֹרִיתֵם אָוְפְּטִימָלִי מְתַסְכֵּל קְדִימָה וְהִוא יְבַחֵר כְּקוּרְבָּן אֶת מֵי שְׁבִשִּׁימָוֹשׁ בְּעַתִּיד הַכִּי רָחֹק, אֲךָ זֶה רָק אָלגוֹרִיתֵם שְׁנִיתֵן לְהַשּׂוֹת בְּאַמְצָעָתוֹ אָלגוֹרִיטְמִים אַחֲרִים וְאֵינוֹ נִיתֵן לְיִישּׁוֹם.



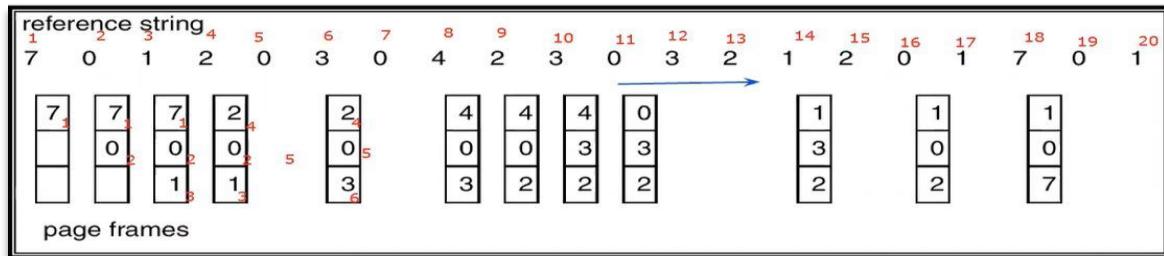
.3. מִנִּיחַ שֶׁאֵם הַשְׁתָּמָשְׁתִּי בְּדָף מְסֻויִּם בְּזָמָן הַאַחֲרָון אֶז הַסְּבִירָות הִיא שָׁאַנְיַ אֲצַטְרֵךְ אֶתְוֹ שָׁוב בְּזָמָן הַקְּרוּב, כְּלֹומֶר הַסִּיכּוֹן שְׁנַצְטַרְךְ דָּף שֶׁלֹּא נְגַעַנוּ בּוּ כָּבֵר שָׁעָה הִיא סְבִירָה פְּחוֹת. לְכָן נְבַחֵר כְּקוּרְבָּן אֶת הַדָּף שְׁנַגְעַנוּ בּוּ בְּעַבְרַ הַכִּי רָחֹק. בְּדָרֶךְ כָּלִיל יוֹתֵר טֻוב מְאָלגוֹרִיתֵם FIFO.

- Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5



- Counter implementation
  - Each page entry has a time-stamp; when a page is referenced, copy the clock into the time-stamp field.
  - When a page needs to be replaced, look at the time-stamps to determine which one should be swapped out.

ניתן למשתמש את זה באמצעות **Time Stamp**, כלומר בכל פעם שיש גישה לדף מסוים אז נעדכן את ה- Time Stamp, כלומר נעדכן את התוויות הזו ליד על כניסה בטבלה דפים, כשנចטרך להחליף אז נחפש את ה- Time Stamp הći נמור.



ניתן למשתמש גם באמצעות תור.

— **Counting Algorithms**

**LFU (Least Frequently Used)** — אם יש דפים שלא ניגשנו אליהם הרבה פעמים אז נרצה לבחור אותם הקרוב.

**MFU (Most Frequently Used)** — ההפך מהגישה של LFU, כלומר נבחר הקרוב את מי שניגשנו אליו הכי הרבה.

— **The Clock Algorithm** – אלגוריתם שמשימוש גם בוינדוס וגם בלינוקס, אלגוריתם מוצלח בכך שהוא פשוט להפעלה.

נסדר את הרשימה של המסגרות שקיימות במערכת שיכולה להחזיק דפים באיזשהו סידור מעגלי למשל באפר מעגלי של מספרים בצורה של שעון, האלגוריתם עובר על המסגרות מהם פוטנציאלי לפינוי והוא ימצא הקרוב לפינוי.

לכל מסגרת צו באפר נשמר איזהו בית שנקרא **Access Reference Bit** או **Bit**, הביא זהה אומר שניגשו לנתונים בזיכרון שם בזמן האחרון, כלומר כל פעם שיש גישה לזכרון למסגרת מסוימת אז הבית הזה נדלק. **מי שמדליק את הבית הזה הוא-MMU.**

נאמר שמערכת הפעלה החליטה שצריך לפנות מקום בזיכרון, כלומר לעשות **Page Out** אז צריך לבחור הקרוב, מערכת הפעלה משתמשת על הדף שמצובע על ידי המחוג של השעון, היא רואה שה- Access Bit דלק, אז היא מכבה את ה- Access Bit ומzieה את המחוג מסגרת אחת קדימה, ה- Access Bit מהווה בעצם כמו כרטיס יציאה חד פעמי מהכלי.

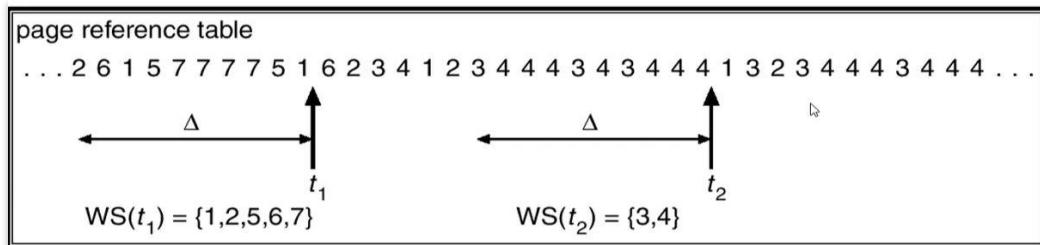
נכתב על ידי אוחד אمسلم, תשפ"ב.

כאשר הוא מגיע למסגרת שה- Access Bit שלו כבוי אז הוא יהיה הקורבן  
ואוֹתָה מסגרת תפונה לדיסק.

זמן ריצה בזמן הגרוע הוא (n)O ובמקרה הטוב הוא (1)O.

**Thrashing** – אם אנו מרכיבים הרבה תוכניות שצורךן הרבה זיכרון והם צריכים יותר זיכרון מהזיכרון הפיזי הפנוי שיש לנו כרגע ב- RAM אז כל הזמן יהיה Page Fault וזה יגרום לנו לבחירת קורבנות לא טובים. זה גורם לניצול לא עיל של שימוש ב-CPU.

**Working Set Model** – מספר הדפים השונים שהייה גישה לזכרון שלהם במספר פעולות קבוע מסויים שהוגדר מראש, מספר זה נקרא Working Set.



**Page Size Selection** – אמרנו ש- 4KB זו בחירה טובה בגלל שהמספרים מסתדרים ממש יפה, לעומת 12 סיביות אופטט, 20 סיביות למספר דף ואז בטבלה דו שכבותית יש 10 סיביות ואז יש 1024 כניסה בכל טבלה וזה מסתדר יפה.

אם נבחר גודל דף גדול יותר נסבול מ- Internal Fragmentation.

**Interlock O/O** – לעיתים נרצה לנעול דף מסוים בזכרון כדי לתת להם חסינות מהמנגנון של הפינוי, כאמור לא אפשר להם לעוף ב- Page Out.

נכתב על ידי אוֹהֶד אַמְسָלִם, תשפ"ב.