

**הרצאה 1:**

**1. היסטוריה של מערכות הפעלה<sup>1</sup>:**

- a. 1960 – מערכות Mainframe
- b. 1970 – PLATO
- c. 1980 – התחלת ה PC עם MSDOS ו XEROX
- d. 1990 – LINUX, 95WIN ועוד...

**2. מה זה מערכת הפעלה?<sup>2</sup>:**

- a. **התפקיד הבסיסי ביותר של מערכת הפעלה – לאתחל את החומרה**
- b. **Bootloader/BIOS** – מערכת הפעלה בסיסית, מאתחל את ה BUS (ערוץ תקשורת) מאפשר ל CPU לעבוד מול הזיכרון. (גם סוג של Bare Metal Machine)
- c. מערכת ההפעלה היא Extended Machine and a Resource Manager (יוסבר בהמשך)
- d. מערכת ההפעלה ממוקמת בין האפליקציות שרצות על המחשב לבין המשאבים הפיזיים שלו.
- e. היא מספקת לנו Common Interface ודרייברים לרכיבי חומרה שונים שהמחשב שלנו מורכב מהם בכדי שנוכל לפתח עליהם.
- f. Extended Interface - היא חוסכת לנו את הצורך ללמוד ממש לעבוד מול הרכיבים האלה ומספקת לנו API.
- g. **Extended Machine** – מספר תכונות של מערכת הפעלה לרבות:
  - i. ניהול משאבים
  - ii. יציבות ואמינות (מגיב תמיד אותו דבר)
  - iii. Portable - יכולת ריצה על מספר סוגים של מכשירים
  - iv. בטוח ועוד..
- h. **Resource Manager** – מנהלת לנו את כלל המשאבים שמחוברים למחשב, מספר תהליכים מקביליים, זיכרון ועוד.
- i. **CPU Management** – (מאפשר את עיקרון המקביליות שיוזכר בהרצאה 2) מנהלת את כלל הכניסות והיציאות למעבד IO ע"י 3 פעילות בסיסיות:
  - i. Computation - חישוב
  - ii. Communication – העברת מידע על ה BUS
  - iii. **Overlapping** – העמסת עבודות, לבצע עבודה אחרת בזמן שהמעבד מחכה/פעולה אחרת נעשית ועוד...
  - iv. **Scheduler** – אחראי בפועל על העמסת הפעולות.
  - v. **I/O Controller** – (יש לו מעבר משל עצמו) אחראי על ניהול המידע שנכתב ויוצא מהדיסק, ה CPU רק מפעיל אותו ומעביר אליו משימות, אחרי שהוא קיבל את ההוראה הוא כותב/קורא לבד.

**3. סוגי זיכרון (מהמהיר והיקר ביותר לאיטי והזול ביותר):**

- a. Register – הכי מעניין, זה הזיכרון המרכזי של המעבד, הכי מהיר והכי יקר, הוגדר כסוג של API של המעבד.
- b. L1 Cache – כנ"ל על המעבד עצמו.
- c. L2... L4
- d. Main Memory – זה ה RAM
- e. Solid State (SSD)
- f. Disk Drive (HDD)

**4. Interrupt and Input Handling:**

- a. למעבד יש העדפה לאינטראפטים, סיגנל, כולם מגיעים מה Interrupt Controller בהתאם לאורך שנכנס, הוא יודע איזו פעולה לעשות. על כל אחד כזה שמגיע, המעבד עוזב את המ שעושה, מטפל בו קודם ואז חוזר לעבודה שלו.

**5. User Space / Kernel Space:**

- a. נמצאים במרחבי זיכרון שונים<sup>3</sup>, לפני שחילקו לתהליכים שונים, ביצעו את החלוקה הנ"ל.
- b. Privileged / Non Privileged – חלוקה למה שיש לו הרשאות לרוץ ולגשת ל KERNEL ולמה שלא.

<sup>1</sup> לא רלוונטי מעבר, דיבר על היסטוריה של מערכות הפעלה בדקות הראשונות

<sup>2</sup> מאתחלות את המחשב ומתחילות ריצה לפני מערכות הפעלה גדולות יותר LINUX ועוד

<sup>3</sup> ב 59:00 נותן את הדוגמא למה זה לא טוב שכלל האפליקציות יהיו על אותו זיכרון, אחרת משתמשים רגילים יהיו יכולים לפנות ל REGISTERS ועוד דברים וליצור התנגשויות מה שיכול לייצר נזק.

c. יש פונקציות שרק ה KERNEL יכול להריץ, אנו נקראים להם ע"י syscall מה User Space. זה ה INTERRUPT היחיד שאפשר להשתמש בו מה User Space. זה יבצע Context Switch ובעצם ישלח INTERRUPT למעבד<sup>4</sup> ע"י שליחת מספר SYSCALL.

## הרצאה 2 :

6. **Instruction Pointer** – מצביע על הפונקציה הנוכחית שהמעבד מבצע.

7. **Context Switch** - למעבד יש תיעדוף גבוהה יותר לטיפול ב-Interrupt Header גם אם הוא כרגע נמצא בטיפול בפונקציה כלשהי. לטובת זה ב CPU יש Instruction Pointer שמקבל את ה Header ועובר אליו. שלבי ביצוע:
- a. שומרים את ה Program Counter, Stack Pointer ועוד.. במחסנית של ה Thread או ב TCB<sup>5</sup> Thread (Control Block).
  - b. עושים Load new program counter to the Interrupt Header.
  - c. Run.
  - d. חוזרים לפעולה הראשונה שביצענו ע"י המצביעים ששמרנו במחסנית.

8. **Multi Programming** / עיקרון ה"מקביליות"<sup>6</sup> / ריצה מקבילית – בעצם עולם התכנות המקבילי, "תהליכון"<sup>7</sup>, "חוט"<sup>8</sup>, "THREAD". עיקרון שמתבסס כולו על Interrupts לעבודה של המעבד. כל תהליכון יקבל Time Sharing.

a. יתרונות:

- i. Utilization Improvements - משפר את השימוש במעבד.
- ii. Pseudo Parallel Services - מראה מצב ש"כאילו" קיימת עבודה מקבילית של שירותים – על מעבד אחד.
- iii. עבודה מקבילית אמיתית מתקיימת על מעבדים שיש להם יותר מליבה אחת.
- iv. משפר זמן תגובה של משימות אינטראקטיביות (דברים שהמשתמש מריץ).

9. **Thread** – בסופו של דבר, "קוד" שזוכר מקום זיכרון ופונקציה להריץ. נכנסים אחד אחרי השני ל Scheduler. הוא מריץ פונקציות שהן Runnable<sup>9</sup>.

a. היעוד שלו הוא לרוץ.

b. **מצבים שונים של Threads**:

- i. New – יצירה של תהליכון חדש
- ii. Ready Queue – נכנס לתור ביצועים של SCHEDULER
- iii. Running – רץ על המעבד
- iv. Blocked – מקבל הוראה לעצור ואז הוא לא רץ, מחכה שיריצו אותו.
- v. Terminated – סיים את כל הפעולות שלו ונמחק.

10. **Processes<sup>10</sup> מבנה נתונים ששומר בתוכו את הפעולות והתוכן שלו** – מקבלת אשליה של זיכרון מלא, Full Memory Space, כך שלא יוכל לפגוע בתהליכים אחרים. דומה מאוד למכונות וירטואליות. כשאומרים שתהליך רץ מתכוונים בעצם לכך שה Thread שלו רץ.

a. **היעוד שלו הוא לבצע הפרדה בין תהליכים שונים.**

b. לכל תהליך יש לפחות חוט אחד (כמעט תמיד יהיו יותר).

c. יש לו מרחב כתובות משלו, Full Virtual Memory Space.

d. Stack – המחסנים של החוטים שלו.

e. Heap.

f. מידע – DATA

g. Code Segment

h. File/Socket descriptor – פנייה לתקשורת ומערכת הקבצים.

i. Signals – Settings

j. CMD arguments

k. Environment Variables

<sup>4</sup>שיטה שקיימת משנות ה-50, ציין שירחיב הרבה יותר בהרצאה 2

<sup>5</sup> הסבר מתחיל ב 16:00 - [https://en.wikipedia.org/wiki/Thread\\_control\\_block](https://en.wikipedia.org/wiki/Thread_control_block)

<sup>6</sup> הוזכר גם בהמצאה הקודמת

<sup>7</sup> נקבע ע"י האקדמיה ללשון, לא משתמשים בזה.

<sup>8</sup> המונח המקובל ע"י האקדמיה לשימוש היום בישראל.

<sup>9</sup> כלומר קוד שאפשר להריץ אותו ע"י CONTEXT SWITCH, SCHEDULER

<sup>10</sup> מתחיל אחרי ההפסקה – 1:24:00 נותן דוגמה ל CRASH שנגרם בשני תהליכים ע"י בעיה באחד מהם זה מה שהביא לחלוקה של ה

SPACE

- .l PID – Process ID
- .m PPID
- .n Permissions / Privileges

### הרצאה 13:

#### 11. System Calls:

- a. **Fork() – מייצר תהליך חדש**<sup>12</sup>
  - i. יוצר תהליך חדש שהוא העתק של האב (התהליך שקורא ל FORK) כולל כל הרגיסטרים של הקבצים, מפות זיכרון וכו'.
  - ii. השוני היחיד הוא ה PID ו PPID (Parent Process ID).
  - iii. הפונקציה מחזירה:
    - (1) את ה PID של הילד אם נקרא ע"י האב.
    - (2) 0 אם נקרא ע"י הילד.
    - (3) 1- אם נכשל<sup>13</sup>.
  - iv. עיקרון **Copy On Write**<sup>14</sup> במקום לייצר הכל מחדש נעתיק את התהליך הקודם ונעבוד עליו. כלומר, עד שהתהליך לא מגדיר לעצמו זיכרון שונה או תוכן שונה הוא עדיין ירוץ עם מה שהוגדר והוקצה לאבא שלו.
- b. **Getppid()** – פונקציה שמחזירה את התהליך אב של הילד.
- c. **Exec()** – משפחה של פונקציות שמריצות תהליכים, מחליפות בין תהליכים.
- d. **Wait()** – מחכה לשינוי במצב של אחד מהילדים של התהליך. אחרי שהוא מסיים לאסוף את הסטאטוס, אחרי שהוא מסיים, הוא מוסר מהתהליכים שרצים.
- e. **Waitpid() / waited()** – יותר ספציפי מהפונ' הקודמת, מחכה לפעולה ועדכון מתהליך ספציפי ע"י שימוש ב ID שלו.
- f. **Exec()** – משפחה של פקודות שמריצות פקודה מסוימת, מחליף את ה image של התהליך הנוכחי באחד חדש, באותה פקודה שקיבל קודם.
- g. **Errno** – משתנה סביבה של המערכת שמטפל בשגיאות בזמן ריצה, מסמן לנו בד"כ מה נדפק בדרך.
- h. **Zombie** – כשתהליך מסתיים מקצים מחדש את המשאבים שלו, האב אחראי על איסוף סטאטוס הסיום של התהליך ברגע שהוא יוצא. אם הוא לא עושה את זה אז התהליך ילד הופך לזומבי. כלומר, תהליך שסיים והאב לא אסף את הסטאטוס שלו<sup>15</sup>.
- i. **Orphan** – מקרה הפוך שהאב "מת" לפני שהילד מסיים. כלל היתומים מאומצים ע"י התהליך init.

#### 12. Thread / חוט –

##### a. יתרונות:

- i. ריצה במקבילה
- ii. שיתוף של קבצים בין חוטים שונים
- iii. שיפור אינטראקטיביות

##### b. הבדלים מול Process:

- i. **בשורה התחתונה – THREAD נועד לריצה ותהליך הוא מעטפת שלמה שכוללת גם יכולות ריצה.**
- ii. מידע משותף בשונה ממידע ייחודי לתהליך.
- iii. כנ"ל בקוד
- iv. כנ"ל בנוגע ל O/I
- v. כנ"ל בנוגע לטבלת סיגנלים.
- vi. לשניהם יש STACK משלהם, כלומר אותו Memory Segment<sup>16</sup>
- vii. כנ"ל בנוגע ל PC
- viii. כנ"ל בנוגע ל REGISTERS
- ix. כנ"ל בנוגע ל STATE
- x. מעבר Context Switch בין חוטים הוא זול יחסית לתהליכים שם הוא כבד יותר.

##### c. **User Level / Kernel Threads**<sup>17</sup> –

- i. (שאלת ראיון עבודה) ב-UNIX/LINUX - יש שני Scheduler שונים שמנהלים את התיעדוף ביניהם, ב KERNEL מה שמריץ אותם הוא של מערכת ההפעלה וב USER SPACE יש User Level Scheduler.

<sup>11</sup> בתחילת ההרצאה עושה חזרה על מה שנלמד עד כה בדגש על תהליכים, Interrupts, חוטים ועוד... עד 20:00

<sup>12</sup> קוד לדוג' ב 24:20

<sup>13</sup> יכול לקרות אם יש יותר מדי תהליכים, אין משאבים ועוד..

<sup>14</sup> דוגמא ב 34:00

<sup>15</sup> במערכות UNIX חדשות זה לא יקרה כי יש תהליך "אימוץ"

<sup>16</sup> נקרא גם safe storage / safe segment (TLS/TSS)

<sup>17</sup> מתחיל ב 1:26:00

d. **POSIX<sup>18</sup>** – ספריה שמייצרת ומנהלת חוטים, לא כל מערכת תומכת בזה (כגון WINDOWS)

**int pthread\_join (pthread\_t th, void\*\* thread\_return)**

Suspends the execution of the calling thread until the thread identified by *th* terminates.

On success, the return value of *th* is stored in the location pointed by *thread\_return*, and a 0 is returned. On error, a non-zero error code is returned.

At most one thread can wait for the termination of a given thread. Calling **pthread\_join** on a thread *th* on which another thread is already waiting for termination returns an error.

*th* is the identifier of the thread that needs to be waited for

*thread\_return* is a pointer to the returned value of the *th* thread (can be NULL).

**void pthread\_exit (void\* ret\_val)**

Terminates the execution of the calling thread. Doesn't terminate the whole process if called from the main function.

If *ret\_val* is not null, then *ret\_val* is saved, and its value is given to the thread which performed *join* on this thread; that is, it will be written to the *thread\_return* parameter in the **pthread\_join** call.

i.

## הרצאה 194:

13. **IPC – Inter Process Communication**: תהליך בסופו של דבר הוא "מקום מבודד" בזיכרון אבל צריכה להיות לו היכולת להעביר מידע בינו לבין תהליכים אחרים.

a. **מוטיבציה:**

i. ידוע של תהליכים חשובים:

(1) שגיאות

(2) בקשות של המשתמש להרוג את התהליך

(3) עצירה על BREAKPOINT למטרות DEBUG

b. כלל התהליכים מנוהלים במנה הנתונים <sup>20</sup>Process Control Block.

c. **דוגמא/בעיה<sup>21</sup>**: מצב שבו תהליך נתקל בבעיה, כגון חלוקה באפס, צריך להודיע לתהליך שהוא נתקבל בבעיה, איך נעביר את המידע/סיגנל הזה לתהליך?

14. **SIGNAL** – יכול להישלח ע"י ה KERNEL או ע"י תהליך אחר. סוגי סיגלים:

- **SIGSEGV – SEGmentation Violation**
- **SIGFPE – Floating point error, eg division by 0**
- **SIGILL – Illegal instruction**
- **SIGINT – Interrupt, eg by user pressing ctrl+C. By default causes the process to terminate.**
- **SIGABRT – Abnormal termination, eg by user pressing ctrl+Q.**
- **SIGTSTP – Suspension of a process, eg by user pressing ctrl+Z**
- **SIGCONT – Causes suspended process to resume execution**
- Which are synchronous?
- **More POSIX signals**

Signals 1,2,3 are synchronous, since they may arrive only as a response to a command that has been executed

a. **טיפול בסיגנלים והתנהגות דיפולטיבית לטיפול בהם:**

i. התנהגות **דיפולטיבית** ABORT, כל עוד לא הגדרנו טיפול אחר בסיגנל SIG-DFLT.

<sup>18</sup> מתחיל 1:33:00 מדבר ארוכות על יצירה של חוטים עד סוף ההרצאה

<sup>19</sup> ב-20:00 הראשונות לא נמצא בשיעור.

<sup>20</sup> [https://en.wikipedia.org/wiki/Process\\_control\\_block](https://en.wikipedia.org/wiki/Process_control_block)

<sup>21</sup> מדגים את הבעיה הנ"ל עד 35:00

- (1) SIGTERM עושה טרמינציה של התהליך.
- (2) floating point exception – SIGFPE
- (3) SIGCHILD – הילד יצא או חוסל, האב מקבל את ההודעה, דיפולטיבית הוא מתעלם או מבצע

#### Default Action

ii. **Signal Handlers** - ניתן להגדיר ידנית את הטיפול בסיגנל (שינוי הערך הדיפולטיבי) ע"י קריאה לפונקציות:

(1) **Signal()** / **sigaction(22)**

- a. מומלץ להיות עקבי בשימוש בפונקציות הנ"ל, לא לקרוא לשניים שונים במהלך התכנות
- b. ההבדלים ביניהם גדולים, הראשונה פשוטה יותר והשנייה יציבה וגמישה יותר. לדוג':

### signal()

**sighandler\_t signal (int signum, sighandler\_t handler)**

- Installs a new signal handler for the signal with number **signum**.
- The signal handler is set to **sighandler** which may be either
  - A user specified function
  - **SIG\_IGN** (ignore the signal)
  - **SIG\_DFL** (use the default signal's actions)
- **signal()** is **one-shot**
  - Should be called again after every signal caught
- Just as bad as one-time dishes

(2) **Sigblockmask()** – משנה את הרשימה של הסיגנלים החסומים.

(3) **Signal Blocking** – אומר למערכת ההפעלה להשהות את הטיפול בסיגנל למועד אחר<sup>23</sup>.

(4) **התעלמות** ע"י **SIG\_IGN**.

iii. **טיפול בסיגנל ע"י USER FUNCTION**, אנחנו מגדירים מה יקרה כאשר ניתקל בסיגנל<sup>24</sup>.

iv. **Default Actions (הכי נפוצים השניים הראשונים):**

- (1) **EXIT** – מאלץ את התהליך לצאת.
- (2) **CORE<sup>25</sup>** - מייצר קובץ **CORE** ויוצא.
- (3) **STOP** – עוצר/משהה את התהליך.
- (4) **IGNORE**
- (5) **CONTINUE**

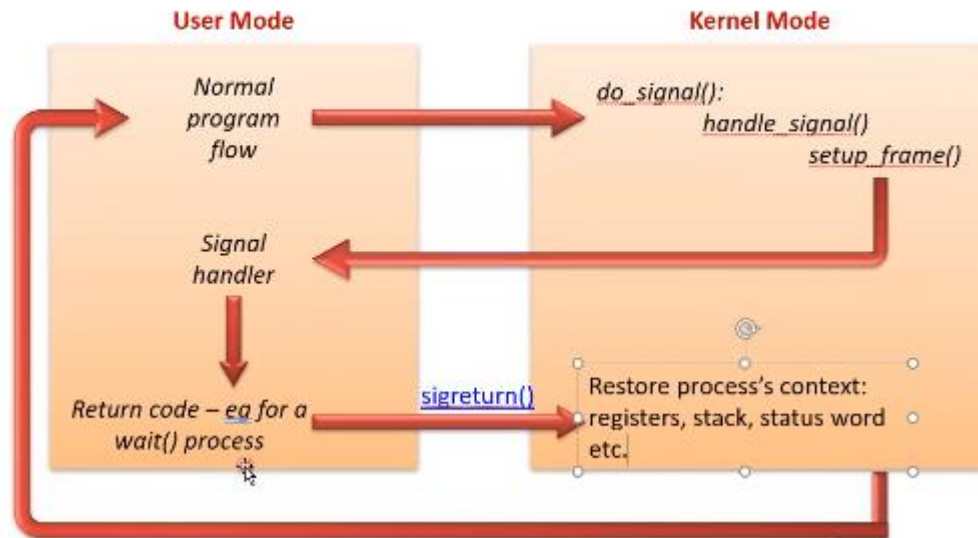
<sup>22</sup> <https://man7.org/linux/man-pages/man2/sigaction.2.html>

<sup>23</sup> [https://www.gnu.org/software/libc/manual/html\\_node/Blocking-Signals.html](https://www.gnu.org/software/libc/manual/html_node/Blocking-Signals.html)

<sup>24</sup> הטיפול עדיין תלוי במערכת ההפעלה, ב **KERNEL**, אם הוא מחליט שהטיפול לא מספיק טוב אוח שצריך לעשות **ABORT** הוא יעשה זאת.

<sup>25</sup> [https://en.wikipedia.org/wiki/Core\\_dump](https://en.wikipedia.org/wiki/Core_dump)

## Signal Processing Scheme



vi. **מגבלות טיפול בסיגנלים :**

- (1) (שאלה מראיון עבודה) לא כל סיגנל ניתן לתפוס, לדוג':
  - a. SIGKILL – ישר הורג את התהליך, לא ניתן להמשיך אחרי.
  - b. SIGSTOP – עוצר את התהליך, יהיה ניתן להמשיך בהמשך.
- (2) כאשר אנו עושים FORK אנחנו מעתיקים גם את ההתנהגות טיפול שהוגדר לתהליך אב, בכדי לאפס את הנייל נצטרך לקרוא לפונ' `execvp()` שמאפסת ומחזירה לדיפולטיביים את הטיפול בסיגנלים.
- (3) **Signal Handler** – לא יכול לראות את המצב של התהליך, הוא מקבל רק מספר טיפול של SIGNAL.

vii. **Real Time Signals :**

- (1) לא מוגדרים מראש, ניתנים להגדרה ע"י האפליקציה. לא כל כך משתמשים בזה בתהעשייה.
- (2) ב POSIX מוגדרים לנו 32 סיגנלים בזמן אמת שהם מורכבים יותר.
  - a. יכול להשהות כמה INSTANCES במקביל.
  - b. מאפשר מידע עשיר יותר.
  - c. עובר בסדר קבוע.

viii. **שליחת סיגנלים בפועל :**

- (1) מהמקלדת
- (2) מה CMD ע"י הפקודה `KILL26` :

```
$kill -SIGTERM <pid>
$kill -1 <pid>
$kill -9 <pid>
```

a. SYSCALL (3)

## הרצאה 5 :

15. **מילון מונחים ומדדים בנושא תיעדוף ותזמון :**

- a. **Throughput** – מס' התהליכים שמסתיימים פר זמן נתון / כל יחידת זמן.
- b. **Efficiency: CPU Utilization** – אחוז הזמן שבו המעבד עסוק (יש לו משימות)<sup>27</sup>.
- c. **Turnaround Time** – הזמן הממוצע בין להגשת המשימה לתור ועד שסיימו.
- d. **Waiting Time** – סכום כל הזמן של כל האינטרבליים שתהליך היה בתור לביצוע.
- e. **Response Time** – הזמן בין הגשה של משימה על ל OUTPUT הראשון שלו.
- f. **Fairness** – תהליכים דומים צריכים לקבל משאבים דומים.

16. **תיעדוף Interrupts (כללי) :**

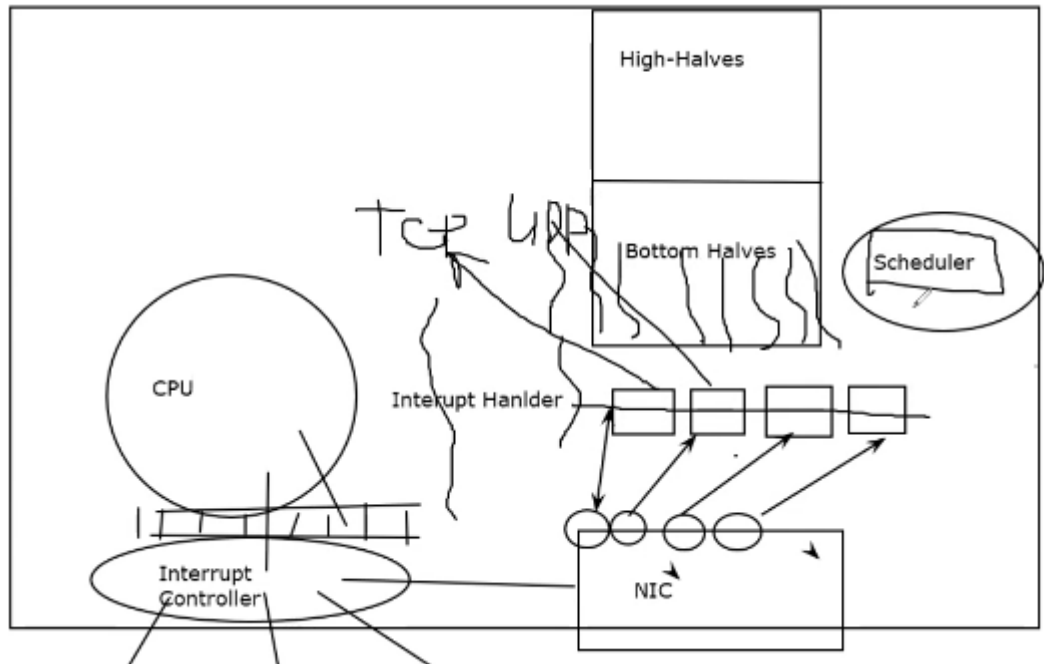
- a. טיפול ב Interrupts מתועדף מאוד גבוהה וחייב להסתיים מאוד מאוד מהר. לדוג': אסור לתת פקודה לישון
- b. חלוקת התעדוף בצורה גסה היא ל חצי עליון וחצי תחתון (HALVES)

<sup>26</sup> <https://man7.org/linux/man-pages/man2/kill.2.html>

<sup>27</sup> זה עקרון המקביליות, כל זמן שיש משהו מחכה, צריך למצוא משימה אחרת לעשות.



- i. בחצי התחתון, Bottom Halves, התיעדוף יעשה ע"י ה Scheduler.  
 c. **Interrupt Handler**<sup>28</sup> – הקוד שירוצ כאשר יקרא אותו האינטראפ.  
 d. **Interrupt Controller** – מס' מקורות Interrupt אפשריים לתהליך הנוכחי ומה הקיבולת שלהם<sup>29</sup>.



#### 17. Scheduler המתזמן:

- a. כל החוטים שבמצב **READY**, מחכים לכו עיבוד, מגיעים למתזמן.  
 b. **CPU Bound / IO Bound** – שני סוגים שונים של משימות שניתן להעביר ל SCHEDULER, האחד שדורש כוח חישוב גבוהה ע"י ה CPU והשני דורש רק IO (כמו הרצת מוזיקה).  
 c. סוגי מערכות ב OS בראיית ה Scheduler:  
 i. **Batch** – בדרך כלל התיעדוף הנמוך ביותר, לא אינטראקטיבי מול המערכת. משתמשים בבנקים, סופרים וכו'.  
 ii. **Interactive** – בעל מטרות רב משמעיות, כבד יותר, בדרך כלל מערכות הפעלה ושרתים.  
 iii. **Real-Time** – משימות שחייבות להסתיים בזמן, התיעדוף הגבוהה ביותר. שעונים חכמים, רכבים אוטונומיים ועוד.  
 d. מטרות המתזמן – תלוי מערכת:  
 i. להיות הוגן עם כל משימה  
 ii. אחראי על אכיפת המדיניות של המערכת הפעלה.  
 iii. אחראי על איזון בין העבודות השונות.  
 במערכות שונות:

#### Batch systems

- Throughput - maximize jobs per hour
- Turnaround time - minimize time between submission and termination
- CPU utilization - keep the CPU busy all the time

#### Interactive systems

- Response time - respond to requests quickly
- Proportionality - meet users' expectations

#### Real-time systems

- Meeting deadlines - avoid losing data
- Predictability - avoid quality degradation in multimedia systems

#### e. סוגי תזמון שונים<sup>30</sup>:

- i. **Preemptive** – תלוי זמן, גם אם לא סיימת מחליפים את החוט בחוט אחר.  
 ii. **Non-Preemptive** – רק ברגע שחוט מסיים עבודה רק אז מקצים למשהו אחר, בין אם נתקעת או החלטת בעצמך.

<sup>28</sup> [https://en.wikipedia.org/wiki/Interrupt\\_handler](https://en.wikipedia.org/wiki/Interrupt_handler)

<sup>29</sup> <https://www.sciencedirect.com/topics/computer-science/interrupt-controller>

<sup>30</sup> <https://www.geeksforgeeks.org/preemptive-and-non-preemptive-scheduling/>

- **Preemptive Scheduling**

A task may be rescheduled to operate at a later time (for example, it may be rescheduled by the scheduler upon the arrival of a "more important" task).  
Pay attention of too many context switches' overhead.

- **Non-Preemptive Scheduling**

Task switching can only be performed with explicitly defined system services, e.g.  
- I/O operation which block the process  
- explicit call to yield()



18. אלגוריתמים לתזמון :

a. **FCFS – First Come First Served**

- i. Non-Preemptive
- ii. הוגן בזמני ההמתנה
- iii. מתאים למערכות BATCH
- iv. לא יעיל בשימוש ב IO

b. **SJF – Shortest Job First**

- i. Non-Preemptive (יש גם הפוך)
- ii. מאפשר זמני turnaround מינימליים.
- iii. **חיסרון** (לא תמיד נדע מראש) צריך לדעת את האורך של המשימה מראש.
- iv. **חיסרון** יכול לגרום ל Starvation<sup>31</sup> (מניעת משאבים מתהליך בצורה שחוזרת על עצמה).
- v. כל המטלות צריכות להיות זמינות מההתחלה בכדי להיות יעיל יותר (כי אחרת יכנס משהו קצר והוא יחכה בתור הרבה זמן).

c. **SRTF – Shortest Remaining Time First**

- i. Non-Preemptive וגם הפוך. זה סוג של הקודם.
- ii. מאפשר זמני turnaround מינימליים.
- iii. מתאים לעבודות אינטראקטיביות.
- iv. צריך לדעת כמה זמן נשאר למשימה.

d. **HRRN – Highest Response Ratio Next**

- i. Non-Preemptive
- ii. מנסה להימנע מהבעיות של SJF ע"י לקיחה בחשבון של הזמן שחיכה התהליך.
- iii. התיעדוף מחושב ע"י נוסחה :

$$Priority = \frac{waiting\ time + estimated\ run\ time}{estimated\ run\ time} = 1 + \frac{waiting\ time}{estimated\ run\ time}$$

e. **Round Robin**

- i. כיום זה האלגוריתם הדיפולטיבי בלינוקס וב UNIX עם Flavor שונים (בנוסף לתיקון קטן שיעלה בהמשך השיעור) בכללי יש לו מעט שכלולים לאלגוריתם הזה.
- ii. Preemptive
- iii. הוגן בחלוקת המשאבים.
- iv. מגדירים Quant לכל משימה ומשימה וכך זה יחולק ע"י Slice זהים של זמן.
- v. לכל המשימות יש תיעדוף זהה.
- vi. מתאים גם לאינטראקטיב וגם ל BATCH.

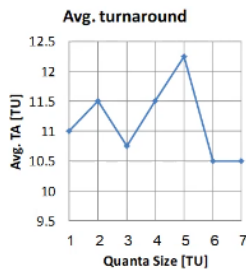
f. **Multi-Level Queue Scheduling**

- i. מה שממומש בפועל היום מערכות ההפעלה Windows, Mac ועוד.
- ii. מחלק את התור הקיים לכמה תורים.
- iii. כאשר תהליך מחכה בתור נכנסים להתליך aging שבמהלכו אנחנו מעלים לו את התעדוף בכדי שלא יורעב.
- iv. באופן דומה, תהליך שמקבל הרבה כוח ותיעדוף על המעבד, אפשר להוריד לו את התיעדוף.
- v. תהליכים אלו נקראים Promotion Up and Promotion Down.

<sup>31</sup>[https://en.wikipedia.org/wiki/Starvation\\_\(computer\\_science\)](https://en.wikipedia.org/wiki/Starvation_(computer_science))



- a. **שאלה:** מערכת מריצה תהליך בודד שמחכה ל IO כ-60% מהזמן בממוצע. מה הניצול CPU המוערך (CPU Utilization)?
- תשובה:** אם תהליך חוסם IO ל-60% מהזמן אזי הניצול של המעבד הוא רק 40% כי בכל שאר הזמן הוא מחכה ואין ניצול.
- b. **שאלה:** אותו המצב רק שהפעם יש 3 תהליכים שרצים
- תשובה:** נשים לב שבכל זמן נתון **ההסתברות**<sup>34</sup> שכל שלושת התהליכים חוסמים IO הוא  $0.6^3 = 0.216$ . כלומר הניצול CPU הוא  $1 - 0.216 = 0.784 \approx 78.4\%$ . זאת מכיוון שפחות סביר ששלושת התהליכים כולם מחכים, אחד מהם כנראה יקבל.
- c. **שאלה:** ישנם מס' תהליכים שדורשים טיפול, מה הזמן QUANTA האופטימלי בטיפול כאשר אני באלגוריתם Round Robin? זאת בהנחה שאנחנו לא משקללים זמן Context Switch אזי כאן נגדיר אותו להיות 0. המטרה שלנו להגיע ל minimal average turnaround time.
- תשובה:** אנו מגלים שלא כדאי לקחת זמני QUANTA נמוכים מדי ועוד אם היה צריך לחשב את ה CS אפילו הממוצע היה ארוך יותר<sup>35</sup>.



$$P_A=6, P_B=3, P_C=1, P_D=7$$

- Quanta = 1:

A	B	C	D	A	B	D	A	B	D	A	D	A	D	A	D	D
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17

$(3+9+15+17)/4 = 11 \text{ TU}$

- Quanta = 2:

A	A	B	B	C	D	D	A	A	B	D	D	A	A	D	D	D
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17

$(5+10+14+17)/4 = 11.5 \text{ TU}$

- Quanta = 3: 10.75 TU
- Quanta = 4: 11.5 TU
- Quanta = 5: 12.25 TU
- Quanta = 6: 10.5 TU
- Quanta = 7: 10.5 TU

- d. **שאלה:** בנושא **Non-Preemptive Scheduling**, 5 משימות מגיעות בערך באותו זמן. מצ"ב התיעדוף שלהם והזמן ריצה המשוער:

PID	Priority	Time
P <sub>1</sub>	3	10
P <sub>2</sub>	5	6
P <sub>3</sub>	2	2
P <sub>4</sub>	1	4
P <sub>5</sub>	4	8

כלל המשימות הן CPU Bound. אנחנו מתעלמים ב CTXW<sup>36</sup>. צריך לחשב לכל אלגוריתם את הזמן TU המינימאלי.

**תשובה:**

- Priority Scheduling (non-preemptive, Higher number means higher priority), 1. Priority Scheduling:  $(6+14+24+26+30)/5=20$
- Non-preemptive FCFS, assuming the jobs arrived in inc. order (P<sub>1</sub>, P<sub>2</sub>, ..., P<sub>5</sub>) 2. FCFS:  $(10+16+18+22+30)/5=19.2$
- Non-preemptive Shortest job first. 3. SJF:  $(2+6+12+20+30)/5=14$

- i. כאשר מדובר ב SJF אנחנו נכניס לפי הקצר ביותר לתור, נשים לב שבכל אחד נסכום גם את הזמן שלו וגם את הזמן של מי שלפניו – ראינו שקיבלנו את התוצאה הטובה ביותר.
- ii. כאשר אנו מתעלמים נטו לפי התיעדוף אנו רואים שקיבלנו את התוצאה הגרועה ביותר.
- e. **שאלה:** בנושא Preemptive Dynamic Priorities, נבחר אלגוריתם לדוגמא שמתעדף כל תהליך לפי מס', גבוהה יותר שווה מתועדף יותר, תהליך מתחיל בתיעדוף 0. כאשר תהליך מחכה למעבד התיעדוף שלו עולה בקצב a וכשהוא רץ התיעדוף משתנה בקצב b. אנחנו יכולים לקבוע את המשתנים הנ"ל.

<sup>32</sup> מתחיל באזור 49:00

<sup>33</sup> אחרי 1:30:00 הוא מראה עוד ועוד דוגמאות מתקדמות ואומר שזה רשות וכדאי לעבור על זה, על כן לא נכנס לסיכום.

<sup>34</sup> בדומה לחוקי ההסתברות ניתן לראות שכאן המשלים של ה IO הוא ה-CPU עצמו ועל כן על הזמן שהם מחכים אנחנו מעלים בשלוש ואז מורידים את התוצאה מ-1.

<sup>35</sup> מעבר לכך, אם הזמן ארוך מדי אנחנו כבר הופכים ל FIFO שזה כבר לא המטרה שלנו במודל.

<sup>36</sup> זמן המתנה של Context Switch בדומה לתרגיל הקודם.

תשובה :

במקרה הבא ניתן לראות שקיבלנו FCFS בגלל שהנתונים גורמים לכך שמי שמגיע קודם יכנס קודם.

- What is the algorithm that results from  $\beta > \alpha > 0$ ?

Consider the following example:  $P_1, P_2, P_3$  arrive one after the other and last for 3 TU.  $\alpha=1$ ,  $\beta=2$  (**bold** marks the running process):

Time	1	2	3	4	5	6	7	8	9
$P_1$	<b>0</b>	<b>2</b>	<b>4</b>						
$P_2$		0	1	<b>2</b>	<b>4</b>	<b>6</b>			
$P_3$			0	1	2	3	<b>4</b>	<b>6</b>	<b>8</b>

The resulting schedule is a non-preemptive **FCFS**.

בדוגמא הבאה קיבלנו LIFO מכיוון שבכל ששן ריצה הזמן ירד אז הוא מחליף ביניהם מהר.

- What is the algorithm that results from  $\alpha < \beta < 0$ ?

Consider an identical example as before, but now  $\alpha=-2$ ,  $\beta=-1$ :

Time	1	2	3	4	5	6	7	8	9
$P_1$	<b>0</b>	-1	-3	-5	-7	-9	-11	-13	-14
$P_2$		<b>0</b>	-1	-3	-5	-7	-8		
$P_3$			<b>0</b>	-1	-2				

The resulting schedule is **LIFO**.

## הרצאה 6 :

### 20. Real Time OS<sup>37</sup> - הגדרה<sup>38</sup> :

- מערכת הפעלה REAL TIME משמעותה שכל משימה מגיעה עם דד-ליין.
- לדוגמא :

- Safety Critical System** – מערכת שלכישלון בה יש משמעות קטסטרופית כגון טילים.
- Hard RT System** – מחייב סיום בדד-ליין אחרת אין משמעות לתוצאות הריצה.
- Soft RT Systems** – כגון מערכות תקשורת, SWITCH, ראוטר רכבים חכמים ועוד. כאן אי עמידה בזמן לא תמיד אומרת שהתוצאה הרסנית או חסרת משמעות.

דוג':

Hard-RT: VxWorks, QNX (Blackberry, Infotainment for Automotive), LynxOS (military), eCos, GreenHills Integrity (avionics)  
Soft-RT: RT-Linux, Windows-CE.

- הערה : מערכת EMBEDDED<sup>39</sup> היא לא בהכרח RT!

### 21. Real Time OS – תכונות והקצאות זיכרון :

- פחות תכונות ומרכיבים מ Server OS/Desktop :
  - חלק מהמערכות הנ"ל הן בעלות מטרה אחת בלבד כגון העברת פאקטות וניתוב, כיוון טיל, GPS וחישוב המסלול הקצר ביותר.
  - אין GUI
  - יכולות חומרה מוגבלות.
- Memory Addressing :
  - לעיתים עובדים עם כתובות פיזיות (בקרנים) – כיום זה נדיר מאוד במערכות הנ"ל.
  - כיום ההקצאות עם וירטואליזציה לרוב.

### 22. Real Time OS – דרישות מימוש :

- Preemptive kernel
- Priority Preemptive scheduler – בהמשך נבחן את קביעת התיעודף.

<sup>37</sup> התחיל בסוף ההרצאה הקודמת, עושה חזרה בתחילת הרצאה 6 וממשיך.

<sup>38</sup> [https://en.wikipedia.org/wiki/Real-time\\_operating\\_system](https://en.wikipedia.org/wiki/Real-time_operating_system)

<sup>39</sup> [https://en.wikipedia.org/wiki/Embedded\\_system](https://en.wikipedia.org/wiki/Embedded_system)

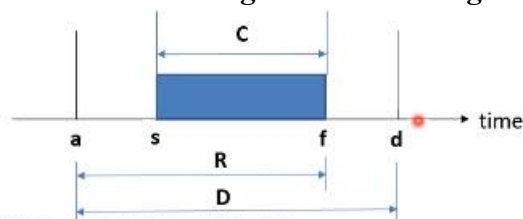
- c. **Low latency** – ברגע שמשימה מתחילה, ישר מגיעה למעבד בלי לחכות הרבה.
- d. **Minimized Jitter**<sup>40</sup> – ממקסם את היכולת לחזות את הזמנים והמשאבים הנדרשים עבור ריצה לדוג' וריאציות של זמני פאקטות ועוד. המטרה היא שעיבוד של כל משימה יהיה ניתן לחיזוי ושהממוצע יהיה אחיד (פיזור מינימאלי).
- e. **מילון מושגים**:
- Event Latency** – זמן מאז שתהליך מתחיל עד שנגמר. לדוג' לחיצה על בלמי הרכב ועד עצירה, לחיצה על העכבר ועוד...
  - Interrupt Latency** – זמן מאז שיInterrupt מגיע עד שהמשימה שלו מתחילה.
  - Dispatch Latency of Scheduler** – זמן שנצרך ע"י ה Scheduler לעצור משימה אחת ולהתחיל אחרת.

## 23. Scheduler – Real Time OS

- המטרה העיקרית היא להגיע לדד-ליין בכלל המשימות אזי הוא לא הוגן בשונה מהאחרים.
- תכונות במערכות הפעלה אחרות שלא רלוונטיות כאן או רלוונטיות פחות:

- Maximum CPU Utilization
- Best Throughput
- Minimum Average Turnaround
- Response and waiting times

## c. אלגוריתם ל Job Timing – RT Scheduling



- a** – arrival (release) time – when job is ready for exec
- d** – absolute deadline – when the job to be completed
- s / f** – when the job starts/finishes
- C** – computation time or *worst case execution time (WCET)* – the time length necessary for CPU to complete the job without interruptions
- R** – response time – the time length since arrival till job finishes:  $(f - a)$
- D** – relative deadline – the time length since arrival till the absolute deadline:  $(d - a)$
- Missing the Deadline: if  $R > D$  or  $f > d$

- D** – הערכת הזמנים אבסולוטית מוגדרת ע"י מהנדסים או תקנים<sup>41</sup>.
- C** – זמן החישוב אנו תמיד ניקח בחשבון את הזמן הגרוע ביותר.

$$U = \sum (C_i / P_i) \text{ - שכלול כלל הפעולות} \quad \text{iii}$$

(1) **C** – זמן COMPUTATION

(2) **P** – זמן אקטיבציה, שווה לסך ה-DEADLINE הרלטיבי.

(3) **U** אם גדול ממספר המעבדים המערכת לא תעמוד בראש בדד-ליין

(4) **לסיכום** – כל משימת זמן אמת מאופיין ע"י שלושה מאפיינים (לפי תעדוף) זמן, דד-ליין וזמן

אקטיבציה. Task (C, D, P)

## 24. סוגי אלגוריתמי תזמון:

a. **EDF – Earliest Deadline First**

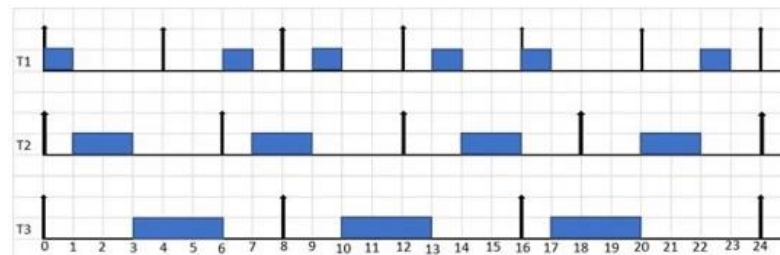
- אין הערכה של P
- Dynamic Priority** – כלומר קובע אותם תוך כדי ריצה ומתאים בהתאם.

<sup>40</sup><https://en.wikipedia.org/wiki/Jitter>

<sup>41</sup> תקנים בדומה לתקנים של פרטוקולי תקשורת בהם לכל אחד זמנים מוגדרים מראש.

iii. לדוג' 42:

- T1 (1,4,4), T2 (2,6,6) and T3 (3,8,8)
- $U = 1/4 + 2/6 + 3/8 = 0.250 + 0.333 + 0.375 = 0.958$  - feasible

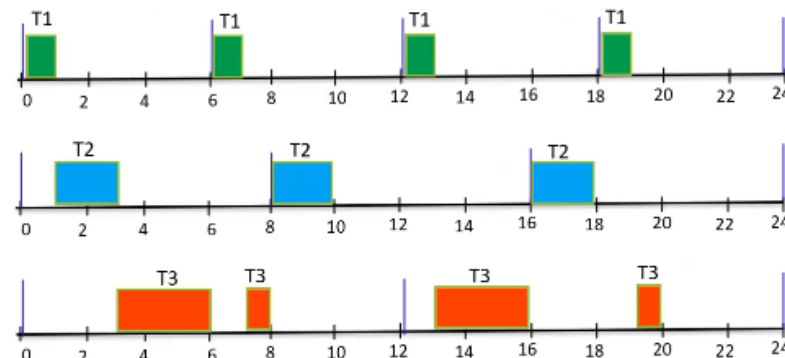


בתמונה הנ"ל T1 יהיה הכי קרוב מכיוון שהדד-ליין שלו הכי נמוך, הוא יכנס ראשון. בכל נקודה ונקודה מסתכלים מתי הדד-ליין הנוכחי של מה שרץ על המעבד ומתי הדד-ליין הכי קרוב של מה שנכנס.

b. **RM – Rate-Monotonic**

- i. מגיע עם Priority קבועים מראש.
- ii. **יתרון** – במקרה ויש כאן עומס יתר המשימות החשובות ביותר יקרו.
- iii. לדוג' :

Example using  $T(C,T,D)$ : T1 = (1,6,6), T2=(2,8,8) and T3(4,12, 12)



בתמונה הנ"ל ניתן לראות שבכל פעם שמגיע T1 בגלל שיש לו את התיעדוף הכי גבוהה הוא יעצור הכל ויכניס אותו למעבד.

c. **יתרונות וחסרונות :**

RM	EDF
Low overhead of scheduling: $O(1)$ with priority sorting in advance	High overhead of scheduling: $O(\log n)$ with AVL tree
For static-priority	For dynamic priority. Optimal
The exact schedulability test is complex, but boundary test is simple	Schedulability test is easy ( $D == T$ )
Least upper bound of U: 0.693	Least upper bound of U: 1.0 ( $D == T$ )
In general, requires more preemption.	In general, requires less preemption.
Practice: easy to implement.	Practice: Complex to implement due to dynamic priorities, but there are known industry designs (Linux).
Rather stable. Even if some lower priority tasks fail to meet the deadlines, others still can do it.	Not stable. If a task fails to meet its deadline, the system may fail due to domino effect. Admission control is desired.

25. **התאמה של Linux ל RT :**

- a. במקור לינוקס לא היה מותאם כלל למשימות RT<sup>43</sup>.
- b. אחרי השדרוגים שביצע Molnar בשנות ה-90 Linux התחילה להתאים למשימות Soft Real Time.

<sup>42</sup> מתחיל להתעמק ב 32:00

<sup>43</sup> מספר על ההיסטוריה ואיגו מולנאר ב 43:00

c. סוגי Linux Scheduling בסיסיים (לפני השדרוג):<sup>44</sup>

- **SCHED\_OTHER / SCHED\_NORMAL**
  - Standard Round-Robin time-sharing policy
- **SCHED\_BATCH**
  - Round-Robin. Tasks are assumed to be non-interactive and CPU-bound with default slice of 1.5 sec. Cache-friendly policy.
- **SCHED\_IDLE**
  - Round-Robin with higher slice given to low-priority tasks
- **SCHED\_FIFO**
  - POSIX RT-class. FIFO without time-slicing
- **SCHED\_RR**
  - POSIX RT-class. RR time-slices with preemption

d. **SCHED\_DEADLINE** הוא פוליסי חדש שהגיע בעדכון של Molnar:

- i. מכניס מנגנון EDF
- ii. CBS – Constant Bandwidth Server Scheduling
- 1) EDF יציב
- 2) מונע את אפקט הדומינו שגורם לתקלות.

## 26. Synchronization

- a. **תיאור הבעיה / מוטיבציה**<sup>45</sup> – מתחיל בהדגמה של קוד שמתאר אינקרמנטציה עם חוטים שמראה שהיא לא מצליחה למספר המיועד.
- b. נריץ `objdump -d -S` ונקבל את הקוד אסמבלי של התוכנה ונראה את הבעיה:

```
gVal++;
40091b: 8b 05 6b 07 20 00    mov     0x20076b(%rip),%eax    # 60108c <gVal>
400921: 83 c0 01             add     $0x1,%eax
400924: 89 05 62 07 20 00    mov     %eax,0x200762(%rip)    # 60108c <gVal>
```

בסופו של דבר הפקודה ++ מורכבת מ 3 פקודות שונות באסמבלי וכאשר שני חוטים עובדים על אותו REGISTER אחד עושה MOV ואז מכל סיבה שהיא התור שלו מסתיים ואז השני נכנס ועושה ADD מה שגורם לכך שפעולת האינקרמנטציה של החוט הקודם נדרסת על ידי השני. ועל כן בסופו של דבר במקום לסכום 2 מליון עשינו הרבה פחות.

c. אחרי שעושים **סנכרון**, הפקודה כולה תרוץ בשורה אחת עם **LOCK** מה שימנע את התקלה:

```
40091b: f0 83 05 69 07 20 00 lock addl $0x1,0x200769(%rip) # 60108c <gVal>
```

## הרצאה 7<sup>46</sup>

### 27. Critical Section / Shared Object<sup>48,47</sup> - דיון על פתרונות אפשריים:

- a. CS הוא זיכרון או כל רכיב כזה ואחר שהוא קריטי לריצה של התכנית.
- b. **תנאים למציאת פתרון טוב:**
- i. **Mutual Exclusion**<sup>49</sup> – נעילה של הקוד, לא יכולים להיות 2 חוטים ב CS באותו זמן, רק אחד אחרת הם פוגעים אחד לשני בעבודה.
  - ii. **Deadlock Freedom** – אם שני תהליכים או יותר מנסים להיכנס ל CS לפחות אחד יכנס כל עוד אין אף אחד להיכנס.
  - iii. **Starvation Freedom** – אם תהליך מנסה להיכנס אז בסופו של דבר הוא יכנס<sup>50</sup>.
  - iv. **Unnecessary Waiting Freedom** – שום תהליך שמחכה להיכנס לתור צריך למנוע מתהליכים אחרים להיכנס כלומר אין תור קלאסי.
  - v. **Logic Solution** – הפתרון צריך להיות לוגי ולא תלוי חומרה.
- c. **פתרון 1 – Strict Alternation** (נאיבי):

<sup>44</sup> ניתן לשינוי ע"י המשתמש עם הרשאות גבוהות.

<sup>45</sup> מהחזרה מהפסקה, אזור 1:05:00

<sup>46</sup> מתחיל את ההרצאה בהמשך של הנושא מסוף הרצאה הקודמת ומעלה את הדיון של הפתרון.

<sup>47</sup> <https://www.javatpoint.com/os-critical-section>

<sup>48</sup> מתחיל ב 1:25:10 [https://en.wikipedia.org/wiki/Mutual\\_exclusion#:~:text=In%20computer%20science%2C%20mutual%20exclusion,purpose%20of%20preventing%20race%20conditions.&text=This%20problem%20\(called%20a%20race,of%20the%20list%20cannot%20occur](https://en.wikipedia.org/wiki/Mutual_exclusion#:~:text=In%20computer%20science%2C%20mutual%20exclusion,purpose%20of%20preventing%20race%20conditions.&text=This%20problem%20(called%20a%20race,of%20the%20list%20cannot%20occur)

<sup>49</sup> מתחיל ב 1:25:10

<sup>50</sup> זה קצת טריקי כי בסופו של דבר מה זה הזמן הסביר שהתהליך אמור להיכנס בו?



- i. רץ על תהליך עד שהוא מסיים, זה מפר את עקרון ה Deadlock בגלל שהוא דורש לפחות 2 תהליכים בגלל שהם תלויים אחד בשני.
- ii. יוצר Priority Inversion<sup>51</sup>.
- d. פתרון 2 – Sleep & Wakeup – Peterson's Algorithm<sup>52</sup> :
- i. כאן כל תהליך מציין אם הוא מעוניין להיכנס או לא ואחרי שהוא מסיים הוא מוריד את "העניין" שלו להיכנס לביצוע, פותר את הבעיה הקודמת.
- המחשה :

Process 0

```
int interested[2];
interested[0]=interested[1]=FALSE;
int turn;

void enter_region(int process){
    int other = 1;
    interested[0] = TRUE;
    turn = 0;
    while (turn == 0 &&
           interested[1] == TRUE);
}

void leave_region(int process){
    interested[0] = FALSE;
}
```

Process 1

```
int interested[2];
interested[0]=interested[1]=FALSE;
int turn;

void enter_region(int process){
    int other = 0;
    interested[1] = TRUE;
    turn = 1;
    while (turn == 1 &&
           interested[0] == TRUE);
}

void leave_region(int process){
    interested[1] = FALSE;
}
```

- ii. מציג תרגיל עצמי על הקוד ב 39:00
- e. פתרון 3 – Bakery's Algorithm – לא בחומר

## 28. מבוא למעבדים מרובי ליבות – Multi-Core Systems :

### a. Cache-coherent and Non-coherent systems<sup>53</sup> :

- i. קוהרנטי – בגישה של ה-Cache יש תהליך סנכרון והם רואים את אותו התור. בעקבות זאת, יקר יותר.
- ii. לא קוהרנטי – כל אחד על תור אחר.
- b. Test-and-Set Lock<sup>54</sup> :
- i. הפעולה הנ"ל קודם מסנכרנת ערך מול כלל המעבדים ומעדכנת אותו בכדי שכולם יראו אם עלה או ירד בהתאם.
- ii. לכל חוט מחכים שיכנס ויסיים את הפעילות שלו. הוא משנה את הערך עד שהוא מגיע ל-0.
- iii. שיטה זאת לא מבטיחה לנו מניעה של הרעבת תהליכים.

### Test-and-set(value)

do atomically  
prev:=value  
value:=1  
return prev

init: value:=0

### For each thread, do:

1. await test-and-set(value) = 0
2. Critical Section
3. v:=0

- iv. סיכום התהליך - מבצעים קוד של CS אם עוברים Spin Lock עושים את ה CS ואז בסוף עושים Spin Unlock. בסוף כשהוא מסיים הוא עושה סנכרון נוסף.

## הרצאה 8<sup>55</sup> :

### 29. Semaphore הגדרה<sup>56</sup> :

ככלל Semaphore הוא אובייקט, General Synchronization Object, שמאפשר גישה למשאבים לתהליכים שרצים. ניתן לחשוב עליו כעל אישור כניסה למתקן עבודה בו יכול להיות איש אחד בלבד. כאשר תהליך מקבל אותו הוא מחזיק ביכולת לרוץ והוא צריך לשחרר אותו בכדי שהאחרים יוכלו גם להיכנס.

### a. סמנטיקה של Semaphores –

init credits + num of up()'s = num of threads to pass down() without sleep

<sup>51</sup> יוסבר בהמשך הקורס.

<sup>52</sup> [https://en.wikipedia.org/wiki/Peterson%27s\\_algorithm](https://en.wikipedia.org/wiki/Peterson%27s_algorithm)

<sup>53</sup> [https://en.wikipedia.org/wiki/Cache\\_coherence](https://en.wikipedia.org/wiki/Cache_coherence)

<sup>54</sup> <https://en.wikipedia.org/wiki/Test-and-set>

<sup>55</sup> הדגמות מתחילות מ 17:00

<sup>56</sup> [https://en.wikipedia.org/wiki/Semaphore\\_\(programming\)](https://en.wikipedia.org/wiki/Semaphore_(programming))



### 30. Binary Semaphore<sup>57</sup>:

- a. בעל שני אופרציות אטומיות<sup>58</sup>:
- למטה** – ניסיון לכניסה.
  - למעלה** – משחרר את המשאב שהוא תפס ביציאה לקראת חוט אחר שיוכל להיכנס.
- b. מתחיל בערך 1 (מוכן לכניסה) ואחרי שהוא נכנס ל-CS אנחנו מאפסים את הערך. כאשר הערך מתאפס אנחנו עושים Block Process כלומר מכניסים את החוט למצב שינה.
- c. **חסרונות**:
- ניתן לראות ישר ש-Semaphore כמו שהם לא מונעים הרעבה.
  - חוט מבחוץ יכול לעשות UP ולשחרר את המשאב מבלי שהוא בפנים בכלל. זה גורם לו להיכנס יחד עם אחד אחר ל-CS מה שמפר את עיקרון ה-Mutual Exclusion.
- d. **תיקון**:
- תיקון לבעיה השנייה יכול להיות ע"י הוספת מנגנון של **Thread Ownership** של Mutex. שם כל חוט שנכנס מסמן את עצמו כבעלים של המשאב כך **שרק הוא יוכל לשחרר**.



### 31. Counting Semaphore:

- a. דומה לקודם ההבדל הם:
- הוא מאותחל למספר N כלשהו שהוא מספר התהליכים שיכולים להיות ב-CS במקביל.

### 32. Negative Value Semaphore:

- a. מימוש נוסף לאותו אובייקט, סוג של Counting Semaphore. כאן אנחנו מורידים את הערך לפני שאנחנו בודקים, מה שיוצר מצב שאפשר להגיע לערך שלילי.

### 33. Mutex:

- a. בדומה לקודמים, גם אובייקט שנועד לניהול וסנכרון חוטים.
- b. **תכונות**:
- בעלות על חוט** – רק החוט שנעל יכול לשחרר.
  - תמיכה בכניסה מחדש Reentrancy** – אותו חוט יכול להיכנס ונעול מספר פעמים.
  - רק החוט שעשה init יכול לשחרר** – רק החוט שעשה את הפעולה יכול לעשות Destroy לאחר מכן.
  - תומך בירושת תיעדוף, **Priority Inheritance** ומונע **Priority Inversion**<sup>59</sup>.

### 34. תרגול<sup>60</sup>:

- a. **ננסה לייצר Counting Semaphore ע"י שימוש בבינארי ובמשתנה int**.
- הפתרון האופטימלי הוא של Barz<sup>61</sup>.

### הרצאה 9

<sup>57</sup> <https://microcontrollerslab.com/freertos-binary-semaphore-tasks-interrupt-synchronization-u-arduino/> - עם הדגמות קוד

<sup>58</sup> <https://stackoverflow.com/questions/52196678/what-are-atomic-operations-for-newbies>

<sup>59</sup> ילמד בהמשך הקורס

<sup>60</sup> מתחיל אחרי ההפסקה ב 1:20:30

<sup>61</sup> מסיים לעבור על התרחיש ב 1:37:00, ציין שממליץ לעבור עם המתרגלים אך לא בטוח אם הם פנויים לזה.

### 35. בעיית <sup>62</sup>Producer Consumer :

```

#define      N      100          /* Buffer size */
Mutex       UseQ = 1;          /* access control to CS */
semaphore   empty = N;         /* counts empty buffer slots */
semaphore   full = 0;          /* counts full buffer slots */

void producer(void) {
    int item;
    while(1){
        produce_item(&item);    /* generate something... */
        down(&empty);           /* decrement count of empty */
        down(&UseQ);             /* enter critical section */
        enter_item(item);       /* insert into buffer */
        up(&UseQ);               /* leave critical section */
        up(&full);               /* increment count of full slots */
    }
}

```

- i. **הבעיה** : נניח ויש לנו רשימה/תור של עבודות שאנחנו רוצים לבצע. נרצה לייצר שני Poolים של חוטים אחד של יצרנים (אלה שמייצרים את העבודה, בודקים את מה שמקבלים ובמידה ועומד בתנאי מכניסים לעבודה) ואחד של צרכנים (אלה שמעבדים את העבודות בפועל).
- ii. **גודל התור N** – חייבים להגביל את מס' המשימות בכדי שהיצרנים לא יכניסו יותר משימות ממה שאפשר להכיל.
- iii. **Mutex** – אחרי על הנעילה והשחרור של המשימה שנכנסת.
- iv. **Empty / Full** – המשתנים שאחראים לספור כמה משימות נכנסו וכמה יש בתור בכדי לא לחרוג מ N.
- v. **הערה** : תסריט בעייתי מאוד הוא כזה שאנחנו נקצה Thread פר בקשה. זה פתרון גרוע מאוד "אסון" שיגרום לעומס גבוהה על המערכת מכיוון ואין הגבלה על מספר החוטים כלל. על כן **חייבים פתרון עם הגבלה**.
- vi. **נק' למחשבה בבעיה<sup>63</sup>** :
  - (1) מה קורה במידה ואין בכלל משימות רצות?
  - (2) מה קורה ועם המשימות מלאות?
  - (3) מה קורה עם שני תהליכים מנסים להכניס לאותה הרשימה במקביל? לכן צריך להגדיר גם את מבנה הנתונים.

### 36. <sup>64</sup>Deadlocks :

- a. **הגדרה** : Deadlock נוצר כאשר חוט אחד נכנס למצב המתנה בגלל שמשאב שהוא כרגע דורש ומחכה לו נמצא בשימוש ע"י חוט אחר שכרגע גם במצב המתנה.
- b. **תנאים שיכולים לייצר Deadlock (שאלה מראיונות עבודה)** :
  - i. **Mutual Exclusion** – המשאב נשלט ע"י תהליך אחד בלבד בזמן נתון.
  - ii. **Hold and wait** – תהליך יכול לבקש משאב בזמן שהוא מחזיק משאב אחר.
  - iii. **No preemption** – ברגע שקיבלת משאב, אף אחד לא יכול להעיף אותך ממנו, רק אתה משחרר.
  - iv. **Circular wait** – שניים או יותר תהליכים מחכים למשאבים שנתפסים ע"י תהליכים אחרים.
  - v. **נק' למחשבה לפתרון** : לא ריאלי כלל למנוע את שלושת הראשונים, הדבר היחיד שאנחנו יכולים לנסות לפתור הוא את ה Circular Wait.
- c. **כיצד פותרים Deadlock?**
  - i. צריך לוודא שתמיד אחד מארבעת התנאים לא מתקיים, אמרנו כבר שהכי ריאלי הוא האחרון.
  - ii. צריך להקצות משאבים כך שלא ייווצר מצב שהם לא בטוחים.
  - iii. צריך לייצר מעגליות של תהליכים ומשאבים כך שנפנה תמיד משאב בזמן מתאים לקראת כניסה. ניתן לעשות זאת ע"י הרגיה של תהליך, פינוי משאבים מתהליך אחר או התחלה מחדש של תהליך (ניתן גם להעביר את התהליך ל Checkpoint מסוים ולא רק להתחלה).

### 37. <sup>65</sup>The Banker's Algorithm :

- a. אלגוריתם למניעת Deadlock.
- b. משתנים באלגוריתם (ווקטורים) :
  - i. **E** – מס' המשאבים שקיימים מכל סוג
  - ii. **P** – מס' משאבים מכל סוג שנמצאים בתהליכי עיבוד ע"י המעבד.
  - iii. **A** – מס' המשאבים הזמינים מכל סוג.

<sup>62</sup> ממשיך מסוף ההרצאה הקודמת.

<sup>63</sup> מסוף הרצאה 8, יכול להיות סוגיות טובות למחשבה גם למבחן

<sup>64</sup> <https://en.wikipedia.org/wiki/Deadlock#:~:text=In%20an%20operating%20system%2C%20a,held%20by%20another%20waitin%20process>

<sup>65</sup> [https://en.wikipedia.org/wiki/Banker%27s\\_algorithm](https://en.wikipedia.org/wiki/Banker%27s_algorithm)

iv. C – מטריצת האלוקציה הנוכחית.

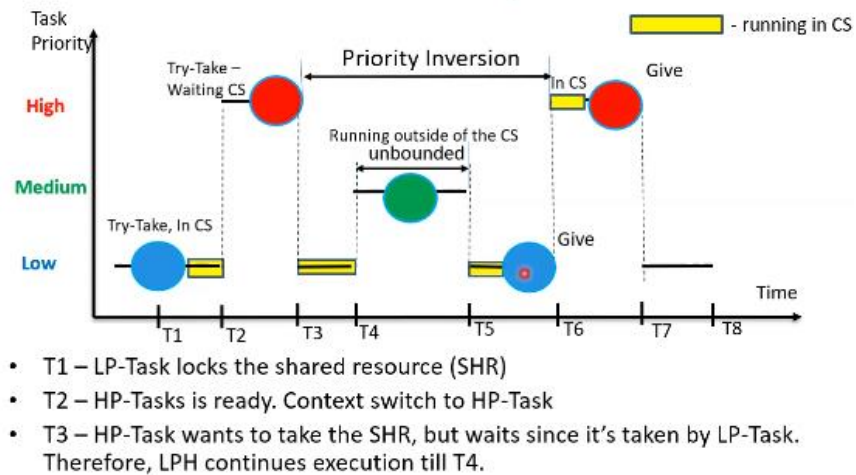
v. R – מטריצת הבקשות.

c. מהלך האלגוריתם:

- נחפש שורה במטריצה R בה צרכי המשאבים הנדרשים קטנים או שווים ל-A. אם לא קיימת שורה כזאת, המערכת יכולה להיכנס ל-Deadlock.
- נניח שכאשר העיבוד של השורה/תהליך שבחרנו תסתיים (סביר) ונסמן את התהליך הזה ככזה שסיים ונוסיף את כל המשאבים שלו ל-A.
- נחזור על צעדים 1 ו-2 עד שכל התהליכים סומנו ככאלה שהסתיימו (מערכת בטוחה)<sup>66</sup>, או עד שנתקלים ב-Deadlock (המערכת לא בטוחה).

38. <sup>67</sup> Priority Inversion – הפיכה/השתלטות על הרשאות<sup>68</sup>:

a. \*\*אחד מההבדלים המרכזיים בין Semaphore ל-Mutex (בשני זה לא נתמך כלל) דוגמא ל-Unbounded Priority Inversion:



בתמונה הנ"ל רואים סיטואציה בה בהתחלה התיעדופים בינוני וגבוהה לא היו צריכים CD (מסיבות כאלו ואחרות, ככה"נ עסוקים במשהו אחר כגון IO) ואז לאור העובדה הנ"ל נכנס התיעדוף הנמוך ל CS והתחיל לעבוד. בגלל שהוא עשה Lock צריך לחכות שיסיים את העבודה שלו ב CS, אזי התיעדוף הגבוהה נכנס להמתנה (Sleep). תוך כדי העבודה של הנמוך עצר אותו התיעדוף הבינוני והתחיל לעבוד. במקרה הזה, מכיוון שהתיעדוף הגבוהה חיכה לנמוך, הבינוני יכול להמשיך לעבוד כמה שרק ירצה בצורה לא מוגבלת (Unbounded) ולא ניתן יהיה לעצור אותו כלל.

b. **Priority Inheritance - הפתרון:** אפשרי רק ב Mutex שבו יש Thread Ownership. במקרה כזה, כשיודעים מי תפס את התור ומי נמצא בהמתנה משווים את התיעדוף של הנמוך יותר ע"י ירושה לתיעדוף הכי גבוהה של מי שמחכה בתור (בעצם משווים ביניהם)<sup>69</sup>. ע"י כך אנחנו בעצם מונעים שתהליכים אחרים עם תיעדוף בינוני ישתלטו באמצע ויתחילו לעבוד ללא הגבלה.

**חיסרון:** למרות שהמנגנון של ירושה פותר את הבעיה של Priority Inversion, הוא לא מבטיח לנו מניעה של **Deadlock!**

<sup>66</sup> מה שאומר שהמערכת היא בטוחה (לא נכנסת ל-Deadlock)

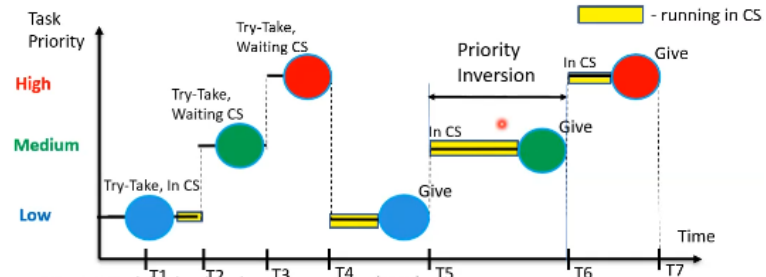
<sup>67</sup> מתחיל 1:30:00 במור"ק על הרכב של NASA במאדים שהיה תקול ולא הצליח לצלם בגלל באג כזה, עוד ניתן למצוא כאן -

<https://www.rapitasystems.com/blog/what-really-happened-software-mars-pathfinder-spacecraft>

<sup>68</sup> [https://en.wikipedia.org/wiki/Priority\\_inversion](https://en.wikipedia.org/wiki/Priority_inversion)

<sup>69</sup> הדבר מתאפשר מכיוון שב Mutex אנחנו יודעים מי מחכה בתור, מה הפרטים שלו ומי נמצא כרגע ב CS

## c. Bounded Priority Inversion



- T1 – LP-Task locks the shared resource (SHR)
- T2 – MP-Task is ready. Context switch to MP-Task
- T3 – MP-Task wants to take the SHR, but waits since it's taken by LP-Task.
- T4 – HP-Task is ready. Context switch to HP-task
- T5 – HP-Task wants to take the SHR, but waits since it's taken by LP-Task.

בתמונה הנ"ל ניתן לראות שאחראי שהמשימה עם התיעדוף הנמוך רצה נכנס מיד אחריה תיעדוף בינוני למרות שיש תיעדוף גבוהה שמחכה בתור. מה שגורם לתופעה הזאת הוא שמשמשים כאן בתור רגיל, ומכיוון שהמשימה הבינונית נכנס לתור אחרי המשימה הנמוכה, היא תצא מהתור לפני המשימה בעלת העדיפות הגבוהה יותר.

**פתרון Priority Queue:** פשוט מאוד נשתמש במבנה נתונים של Priority Queue, מה שיגרום לכך שעצמים בעלי תיעדוף גבוהה יותר יתעוררו קודם.

## 39. תרגול ושאלות חזרה:

a. שאלה: כמה משימות יכולות להיכנס ברגע נתון בקוד הבא?

```
#define N 100 /* Buffer size */
Mutex UseQ = 1; /* access control to CS */
semaphore empty = N; /* counts empty buffer slots */
semaphore full = 0; /* counts full buffer slots */

void producer(void) {
    int item;
    while(1) {
        produce_item(&item); /* generate something... */
        down(&empty); /* decrement count of empty */
        down(&UseQ); /* enter critical section */
        enter_item(item); /* insert into buffer */
        up(&UseQ); /* leave critical section */
        up(&full); /* increment count of full slots */
    }
}
```

**תשובה:** עד 100 בכל רגע נתון, או יצרנים או צרכנים או כל שילוב שלהם שמגיע עד 100.

b. שאלה: מה הבעיה שיכולה להיווצר כתוצאה מהחילוף הבא בין השורות?

```
void producer(void) {
    int item;
    while(1) {
        produce_item(&item); /* generate something... */
        down(&empty); /* decrement count of empty */
        down(&UseQ); /* enter critical section */
        up(&UseQ); /* leave critical section */
        enter_item(item); /* insert into buffer */
        up(&full); /* increment count of full slots */
    }
}
```

No mutual exclusion: addition to buffer is out of "safe code".

**תשובה:** בקוד הנ"ל ייווצר מצב ששני פעולות יכנסו ל CD במקביל, דבר הסותר את עקרון ה Mutual Exclusion, זאת מכיון שההכנה שלנו היא UP וה DOWN, לא יכול להיות שנעשה DOWN ויש UP בלי שהכנסנו לשם את מה שאנחנו רוצים ל-Buffer. דבר זה יוצר סיטואציה שאנחנו תלויים במעבד שעושה את שלושת הפעולות הנ"ל כפעולה אטומית (מה שכמעט תמיד לא נכון) ואנחנו לא יכולים להסתמך בפתרונות שלנו ביכולות החומרה.

c. **שאלה:** מה הבעיה שיכולה להיווצר כתוצאה מחילוף שני השורות הבאות בקוד?

```
void producer(void) {
    int item;
    while(1) {
        produce_item(&item); /* generate something... */
        down(&empty); /* decrement count of empty */
        down(&UseQ); /* enter critical section */
        enter_item(item); /* insert into buffer */
        up(&full); /* increment count of full slots */
        up(&UseQ); /* leave critical section */
    }
}
```

No Problem : Just does non-critical actions in CS.

**תשובה:** אין כאן בקוד בעיה קריטית מה שכן יכול להיווצר מצב שנעשה פעולות לא קריטיות ב CS, אך הקוד יעבוד כרגיל.

d. **שאלה:** מה יקרה אם נבצע את החילוף בקוד הבא?

```
void consumer(void) {
    int item;

    while(TRUE) {
        down(&UseQ); /* enter critical section */
        down(&full); /* decrement count of full */
        remove_item(&item); /* take item from buffer */
        up(&UseQ); /* leave critical section */
        up(&empty); /* update count of empty */
        consume_item(item); /* do something... */
    }
}
```

Deadlock : Empty buffer, consumer blocked at down(&full), producer blocked at down(&UseQ) → both processes sleep.

**תשובה:** כאן יכול להיווצר Deadlock<sup>70</sup>.

e. **שאלה<sup>71</sup> (Deadlock):** נניח ויש לנו מספר מסוים של משאבים שכל אחד מהם הוא ייחודי (כלומר, אין עוד משאבים מאותו סגנון שזמינים). צריך להוכיח שבמקרה והבקשה למשאבים היא לפי סדר עולה<sup>72</sup>, אז במקרה כזה Deadlock הוא בלתי אפשרי.

**תשובה:** אכן בלתי אפשרי מכיוון שכל תהליך מבקש את המשאב אחריו שכרגע תפוס ע"י תהליך אחר, מכיוון שכל תהליך מבקש משאב אחר וייחודי שכבר נמצא אחריו בתור, לא קיים מצב שבו שני התהליכים מחכים למשאבים אחד של השני. כלומר, מנענו כאן את בעיית ה Circular Wait. הדגמה מהמצגת:

- Assume there is a system that has:
  - Processes  $P_1$  and  $P_2$ .
  - Resources  $R_1$  and  $R_2$ .
- Assume that  $P_1$  holds  $R_1$  and  $P_2$  holds  $R_2$ . Now,  $P_1$  requests  $R_2$  and is now waiting for  $P_2$  to release it.
- In order to have a deadlock in the system,  $P_2$  needs to ask for  $R_1$ . However, this contradicts the assumption that resources can only be requested in ascending order.

Which one of the conditions is prevented?  
Condition 4 : Circular Wait

f. **שאלה (Banker's Algorithm):** מה קורה ברגע שמריצים את אותו התרחיש בסדר הפוך?

**תשובה:** לא משנה, עדיין לא אפשרי מאותה הסיבה.

g. **שאלה:** נניח שקיבלנו את הסיטואציה הבאה, האם המערכת נמצאת ו/או תכנס ל-Deadlock?  
למצב הבא<sup>73</sup> נגיע אחרי שנמצא את המטריצה הימנית (בצבע אדום) זאת מטריצת הצרכים של כל תהליך לפי משאב, אנחנו נבנה את המטריצה ע"י חיסור של הצורך בהקצאה הנוכחית (זה בפועל מה שהתהליך עוד צריך).

<sup>70</sup> לא מסביר מעבר, משאיר את זה כחומר מחשבה – יכול להיות תרגול טוב למבחן

<sup>71</sup> חלק שני של שאלות בהרצאה הזאת (אחרי ה Deadlock), מתחיל ב 53:50

<sup>72</sup> כלומר תהליך כלשהו יכול לבקש את משאב 2 אם הוא מחזיק במשאב 1 וכן הלאה

<sup>73</sup> בפועל החלק האדום הוא חלק מהתשובה.



A=	R <sub>1</sub>	R <sub>2</sub>	R <sub>3</sub>	R <sub>4</sub>
	2	1	0	0

Process	current allocation				max demand				still needs			
	R <sub>1</sub>	R <sub>2</sub>	R <sub>3</sub>	R <sub>4</sub>	R <sub>1</sub>	R <sub>2</sub>	R <sub>3</sub>	R <sub>4</sub>	R <sub>1</sub>	R <sub>2</sub>	R <sub>3</sub>	R <sub>4</sub>
P <sub>1</sub>	0	0	1	2	0	0	1	2	0	0	0	0
P <sub>2</sub>	2	0	0	0	2	7	5	0	0	7	5	0
P <sub>3</sub>	0	0	3	4	6	6	5	6	6	6	2	2
P <sub>4</sub>	2	3	5	4	4	3	5	6	2	0	0	2
P <sub>5</sub>	0	3	3	2	0	6	5	2	0	3	2	0

**תשובה:** נחשב את הכניסה והיציאה של תהליכים לפי הסדר הבא ונראה בסופו של דבר שהריצה היא בטוחה (Safe), אנו רואים שכל תהליך שמסיים מפנה מספיק משאבים לתהליך אחר שנכנס אחריו (לא בהכרח לפי הסדר) מה שיוצר מצב שבסופו של דבר אנו מצליחים לסיים את כל המשימות של המערכת.

A=	R <sub>1</sub>	R <sub>2</sub>	R <sub>3</sub>	R <sub>4</sub>
	2	1	0	0

A=	R <sub>1</sub>	R <sub>2</sub>	R <sub>3</sub>	R <sub>4</sub>
	2	1	1	2

A=	R <sub>1</sub>	R <sub>2</sub>	R <sub>3</sub>	R <sub>4</sub>
	4	4	6	6

A=	R <sub>1</sub>	R <sub>2</sub>	R <sub>3</sub>	R <sub>4</sub>
	4	7	9	8

A=	R <sub>1</sub>	R <sub>2</sub>	R <sub>3</sub>	R <sub>4</sub>
	6	7	9	8

A=	R <sub>1</sub>	R <sub>2</sub>	R <sub>3</sub>	R <sub>4</sub>
	6	7	12	12

Process	current allocation				still needs			
	R <sub>1</sub>	R <sub>2</sub>	R <sub>3</sub>	R <sub>4</sub>	R <sub>1</sub>	R <sub>2</sub>	R <sub>3</sub>	R <sub>4</sub>
P <sub>1</sub>	0	0	1	2	0	0	0	0
P <sub>2</sub>	2	0	0	0	0	7	5	0
P <sub>3</sub>	0	0	3	4	6	6	2	2
P <sub>4</sub>	2	3	5	4	2	0	0	2
P <sub>5</sub>	0	3	3	2	0	3	2	0

This is the total amount of resources of the system.

h. **שאלה<sup>74</sup>:** שאלה נוספת שרוברט דילג עליה והמליץ לתרגול עצמי

If a request for (0, 1, 0, 0) arrives from P<sub>3</sub>, can that request be safely granted immediately? In what state (deadlocked, safe, unsafe) would immediately granting the whole request leave the system? Which processes, if any, are or may become deadlocked if this whole request is granted immediately?

**תשובה:**

**הרצאה 10:**

40. **CPP – Ceiling Priority Protocol**

a. אלגוריתם מאוד תעשייתי ומאוד פופולארי היום, הומצא בשנות ה-90, יש לו שני וריאנטים<sup>75</sup>:

i. **OCPP - Original Ceiling**

ii. **ICPP - Immediate Ceiling**

b. **מושגים ומונחים:**

i. **Ceil** – תקרה של Semaphore, זה התיעדוף של התהליך הכי גבוהה שיש בו.

ii. **Task** – משימה שאנחנו רוצים להריץ, משימה שיכולה לנעול את ה-Semaphore. משימה יכולה לנעול

אם ורק אם התיעדוף שלה **גבוהה ממש** מהתקרה של כל התהליכים שכרגע נעולים ע"י משימות אחרות.

iii. **אם התנאי של המשימה לא מתקיים**, כלומר לא ניתן לנעול לו כרגע לנעול ה-Semaphore, יכול להיות

שישלחו אותו לתור של Semaphore אחר.

iv. המשימה שנועלת עדיין עושים **Priority Boosting<sup>76</sup>**.

<sup>74</sup> 1:28:00 הזכיר כי מדובר בתרגול טוב לעבודה ומבחנים, פירוט השאלות במצגת של רוברט.

<sup>75</sup> כמו כן יש לו עוד מספר שמות וכינויים שלא נרחיב כאן, נמצא בשקף 8 של המצגת.

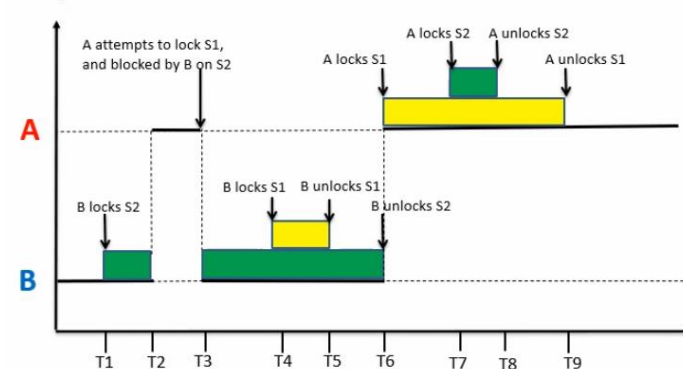
<sup>76</sup> זה התהליך שנבע ישירות מעיקרון הירושה, בכך שהוא יורש את התיעדוף של מי שהיה לפניו ומבטיח את זה שמישהו בתיעדוף נמור יותר לא יוכל להתערב.



Task Name	Time	Priority	Action	Sem Ceiling
A	50	3	lock {S1} lock {S2} ... unlock {S1} unlock {S2}	ceil(S1) - 3 ceil(S2) - 3
B	500	2	lock {S2} lock {S1} ... unlock {S1} unlock {S2}	

בתמונה הנ"ל, אם לא נפעיל את עיקרון ה CPP, אנחנו לא יכולים למנוע Deadlock מכיוון ששני התהליכים דורשים את אותם המשאבים ונועלים את אותם המשאבים, זה מצב בלתי נמנע. אלא אם כן נכניס את עקרון התקרה, שימנע מתהליכים בתיעדוף נמוך יותר לבצע נעילה.

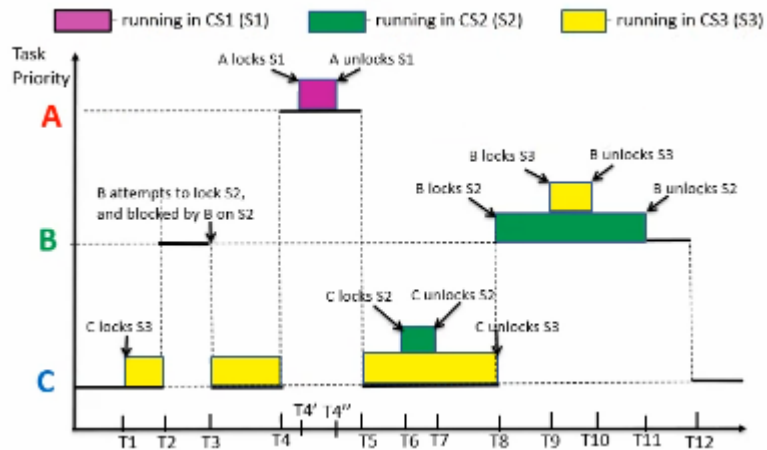
Task   - running in CS1 (S1)   - running in CS2 (S2)



בתמונה הנ"ל ניתן לראות **שנמנע Deadlock** בטוח כתוצאה משימוש בתקרה וזאת מכיוון ש-A לא הצליח להיכנס אחרי ש-B היה בתהליך. מה שכן בין T4 לבין T5 היה Priority Inversion וזאת מכיוון שהתיעדוף של B נמוך יותר אך בכל זאת הוא קיבל משאבים שהגבוהה יותר רצה.

Task Name	Time	Priority	Action	Sem Ceiling
A	50	3	lock {S1} ... unlock {S1} ...	ceil(S1) - 3 ceil(S2) - 2 ceil(S3) - 2
B	500	2	lock {S2} ... lock {S3} ... unlock {S3} ... unlock {S2} ...	
C	3000	1	lock {S3} ... lock {S2} ... unlock {S2} ... unlock {S3} ...	

כאן יש דוגמא דומה לקודמת רק עם 3 משימות, והפעם כל אחת רוצה משאבים שונים. ניתן לראות שערכי התקרה שונים במצב הזה.



גם כאן נוצר לנו מצר של Priority Inversion בין 5 ל-8 אך זה מחיר "נמוך" יחסית שאנחנו משלמים בכדי למנוע Deadlock. כמו כן ניתן לראות שב-B נחסם ע"י S2 מכיוון שהתיעדוף שלו הוא 2 ולא גדול ממש מהתקרה של המשאב שהוא רוצה שהיא גם 2.

לסיכום: e.

i. תכונות מיוחדות:

- (1) OCPP – כאן ה-Boosting מתבצע כאשר משימה אחרת מנסה לנעול את המשאב שהמשימה הנוכחית כבר נעולה עליו, אזי הוא עושה Boost לתקרה של המשאב.
- (2) ICPP – כאן ה-Boosting הוא אוטומטית מקבל את הערך ברגע שהוא נועל משאב, בלי קשר למי שמנסה לנעול במקביל.

ii. יתרונות:

(1) מונע Deadlock.

iii. חסרונות:

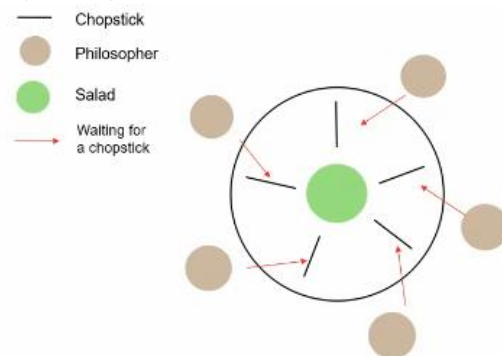
- (1) לא מבטיח מניעה מלאה של Priority Inversion. מה שכן המשימה יכולה להיות "מעוכבת" לכל היותר פעם אחת על ידי משימה עם תיעדוף נמוך יותר<sup>77</sup>.
- (2) ה-Delay שנוצר עקב כך הוא מוגבל, Bounded Priority Inversion.

41. סיכום Priority Inversion ו-RT-Synchronization, מה למדנו עד כה<sup>78</sup>:

- a. קל מאוד ליישם ב-Mutex עיקרון Priority Inheritance.
- b. ב-Semaphore זה קשה הרבה יותר ומצריך במינימום CPP.

42. Dining Philosophers Problem<sup>79</sup>:

- a. תיאור הבעיה: נניח שיש 5 פילוסופים מסביב לשולחן שמנסים לאכול סלט. הם צריכים 2 מקלות אכילה כדי לאכול, כולם מנסים לתפוס את המקל מימין ואז אחרי זה לא יהיה להם משמאל, זה יוצר Deadlock ישר.



- b. פתרון 1: מתבסס על מערך של Semaphores כך שמישהו לא יחכה יותר מדי זמן כי הדבר הזה מבוסס על כך שאחד השכנים יכול ל"הכניס בכוח" את אחד מהם לאכול. זה מונע Starvation Freedom.
- c. פתרון 2 – LR<sup>80</sup>: כאן כל פילוסוף יהיה או ימני או שמאלי. זה פתרון שלא מתבסס על ה-Center Lock. כאן מגיוון שפילוסופים לוקחים ומשחררים את המקלות אזי אין לנו Deadlock מה שמונע גם הרעבה מתהליכים כאלה ואחרים שמנסים לקחת, יכול להיות מקרה שהם יכשלו פעם אחת אבל בסופו של דבר

<sup>77</sup> אנחנו רואים בתרחישים ובהדגמות במהלך השיעור שהאירוע חוזר על עצמו פעם אחת לכל אחד.

<sup>78</sup> מתחיל ב 35:00

<sup>79</sup> גם בעיה שעולה הרבה ברעיונות עבודה.

<sup>80</sup> מבוסס על כך שיש פילוסופים ימניים ושמאליים, מתחיל ב 59:00.

#### 43. זיכרון <sup>81</sup>Memory:

- a. **תיאור הבעיה, מיפוי כתובות וירטואלי<sup>82</sup>:** ברגע שמתכנת כותב תוכנה, נניח והוא מתאים אותה לכתובות הפיזיות שלו בזיכרון, בהנחה שהוא מעביר את זה עכשיו למחשב אחר, הכתובות של הזיכרון לא יהיו אותו דבר כלל. בסופו של דבר כתובות פיזיות של זיכרון תלויות במס' מרכיבים, הגודל שלו, המכשיר עצמו והמעבד. בסופו של דבר לכל מחשב יש מעבד עם ארכיטקטורה שונה ורכיב זיכרון שונה.
- b. **פתרון:** כדי לפתור את הבעיה הנ"ל, הומצא מנגנון בשם **Hardware-Independent Memory Addressing**: המנגנון הזה ממיר את הכתובות הפיזיות של הזיכרון לכתובות וירטואליות, ולכן המתכנת יעבוד על כתובות וירטואליות והקוד עצמו כלל לא יעבוד מול כתובות פיזיות.  

$$\text{Virtual address} = \text{Physical Address} + \text{Normalization Offset}$$
- c. **תיאור הבעיה, יחידות זיכרון <sup>83</sup>Page:** אחרי שעשו חלוקה ל User space/Kernel space ואחרי שהתפתח הקונספט של Multiprogramming, היו צריכים להקצות יותר ויותר זיכרון להרבה מאוד צרכנים במקביל. נניח ויש לנו זיכרון מוגבל ומס' תהליכים שרוצים לנצל את כולו, אם אנחנו נקצה להם זיכרון קבוע לכל אחד ונרצה להחליף ביניהם כל הזמן, תהיה לנו בעיה לאור החלוקה שעשינו והמחיצות ביניהם.
- d. **פתרון <sup>84</sup>Memory Paging:** הפתרון הוא ליצור מנגנון שנותן לכל תהליך יחידות זיכרון קבועות שלא מחולקות בחוצצים כמו המודל הקודם אלא מוקצות בדורה יותר דינאמית במקטעים שלמים. כל אחד מהם יקבל **Memory Space** וכך נוכל לתמרן ביניהם בצורה הרבה יותר נוחה. בשורה התחתונה, כל תהליך היה מקבל תחום כתובות וירטואליות מלא משל עצמו.

#### הרצאה 11:

44. **Memory Segmentation:** כל מנגנון הסגמנטציה נולד בעקבות הרצון להקל כמה שיותר על משאבי העיבוד שהיו יקרים מאוד בתחילת עידן המחשבים. ועל כן רצו שיהיה רצף זיכרון אחיד של Stack ו-Heap. דבר שלא היה מצריך מעבר גבוהה בין כתובות שהיה בזמנו מאוד כבד למחשב.

#### 45. Tradeoffs - Page size / Page-table size:

- a. זיכרון לוגי של 32 ביט בגודל 4GB יכול להתחלק בצורה הבאה לפייגים בהתאם לגודל של כל פייג:
  - i. 1K Page ו-4M דפים.
  - ii. 4K Page ו-1M דפים.
- b. החישוב מבוצע בצורה הבאה בהתאם לגודל שנקבע ל-Page:
 
$$\begin{aligned} \text{Page 16K} &- 4 \text{ bytes/entry} \times 256 \text{ K entries} = 1 \text{ Mb} \\ \text{Page 4K} &- 4 \text{ bytes/entry} \times 1 \text{M entries} = 4 \text{ Mb} \\ \text{Page 1K} &- 4 \text{ bytes/entry} \times 4 \text{M entries} = 16 \text{ Mb} \end{aligned}$$
- c. **מסקנה:** אנו רואים שככל שה-Page גדול יותר, כך הטבלה הסופית (המערך) שאנו צריכים לזיכרון הוא קטן יותר ולהיפך. המצב הזה חוסך לנו מקום ומשאבים לגודל הטבלה אבל זה ההפסד מזה שהוא שהדפים יכולים להיות לא מנוצלים מספיק בגלל הגודל שלהם.

46. **Page Table Consideration:** כיצד ניתן להתמודד עם טבלאות גדולות מדי? ישנם מספר תוכנות ומרכיבים שצריכים להתקיים מבלי קשר לגודל הטבלה. פתרון קיצוני אחד הוא שכל טבלאות המיפוי יהיו בחומרה, זה ככה"נ לא אפשרי מכיוון שאמנם הגישה מאוד מהירה אבל זה יקר מאוד, במיוחד עבור טבלאות גדולות<sup>85</sup>. פתרון קיצוני אחר הוא להחזיק הכל בזיכרון המרכזי (Main memory), זה יהיה מנגנון עם מצביעים ורגיסטרים אבל זה גם יקר מאוד כי אז אנחנו מכפילים כל Reference לזיכרון<sup>86</sup>. מכאן אפשר לבחון כיוון של Paging על ה Page Table עצמו.

<sup>81</sup> מתחיל ב 1:22:00 אחרי ההפסקה.

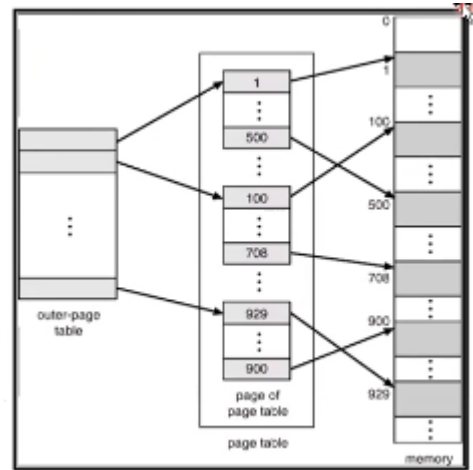
<sup>82</sup> [https://en.wikipedia.org/wiki/Virtual\\_memory](https://en.wikipedia.org/wiki/Virtual_memory)

<sup>83</sup> [https://en.wikipedia.org/wiki/Page\\_\(computer\\_memory\)](https://en.wikipedia.org/wiki/Page_(computer_memory))

<sup>84</sup> [https://en.wikipedia.org/wiki/Memory\\_paging#:~:text=Windows%20uses%20the%20paging%20file,used%20in%20the%20page%20file](https://en.wikipedia.org/wiki/Memory_paging#:~:text=Windows%20uses%20the%20paging%20file,used%20in%20the%20page%20file)

<sup>85</sup> כן ישנם כאלה שכן נחזיק בחומרה, נקרא MMU ויעלה בהמשך הקורס.

<sup>86</sup> בסופו של דבר זה גם הזיכרון וגם מצביע, לא יעיל כלל.



כאן אפשר לראות שאנחנו בעצם עושים מערך חיצוני של PT שמצביע למערך פנימי של PT שמצביע על הזיכרון עצמו. החיסכון כאן הוא שלא כל החלקים בטבלה החיצונית יהיה מאוכלס אלא רק מה שבתפוסה. בדוגמא הראשונה אנחנו נראה כי זה מסייע בכך שבסופו של דבר **רוב הזמן** תהליך לא צריך את כל הזיכרון הוירטואלי שלו.

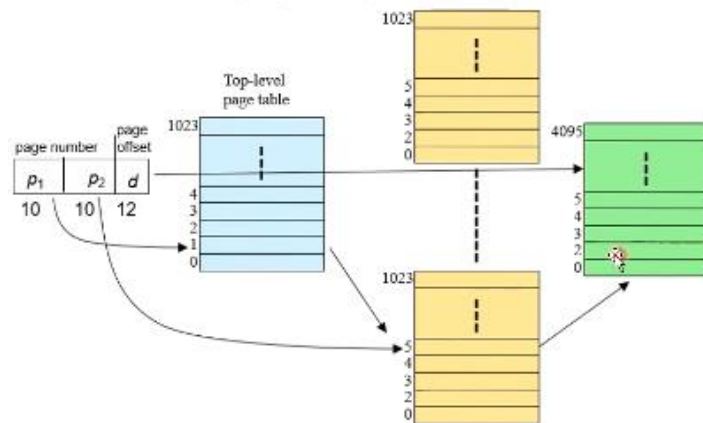
a. **דוגמא 1:** יש לנו מכונת 32 ביט עם 4K גודל Page. אזי החישוב יהיה:  
 $0-4095 = 4K - 2^{12}$  - **החישוב** הוא בצורה הזאת מכיוון ש  $2^{10}$  הוא 1K ואז כפול 4 יוצר בחזקת 12. מכאן נובע ישירות שאנו צריכים תצוגת Page בצורה הבאה:

page number		page offset
$p_1$	$p_2$	$d$
10	10	12

כאן ה Offset מחושב ע"פ הנוסחה הנ"ל. כמו כן, מכיוון שה Page Table עצמו מחולק ל Page אזי תהיה לו חלוקה של המס' Page (ה-20 ביטים הראשונים). כמו כן 1P ו-2P מייצגים את הטבלה החיצונית והפנימית בהתאמה.

**כעת, נניח** שתהליך באותו מכונה משתמש ב 4MB Stack, 4MB Code segment and 4MB Heap (כל אחד כזה צריך אלף עמודים, כל אחד של 4K).

נחשב, ונראה שאנחנו נצטרך 12MB של זיכרון סה"כ לתהליך, 3 טבלאות פנימיות לכל אחד מרכיבי זיכרון השונים של המערכת וטבלה אחת חיצונית שתצביע אליהם (שלא תמיד תהיה בשימוש בהכרח).  
**הטעות בדוגמא 1:** החישוב הנ"ל הוא בהנחה שכל רכיבי הזיכרון **רצופים!** זה לא בהכרח נכון ל Heap ובהכרח לא נכון ל-Code Segment. ועל כן, יש מצב שבמקום דף אחד נצטרך 4 דפים לאחד מהם.



b. **דוגמא 2:** מה יקרה כשאותה כתובת תהיה שוב ושוב בשימוש?

#### 48. 88 89 Inverted Page Table :

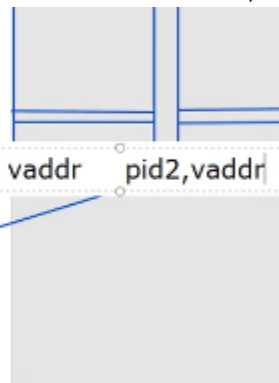
a. **הבעיה:** אנו רואים שהפתרון הנ"ל לא מתאים **למערכת של 64 ביט**. מכיוון שבמערכת כזאת אנחנו נצטרך להקצות יותר מדי זיכרון פר טבלה, לעיתים עד מצב של 4 שכבות של טבלאות. זאת סיטואציה שיכולה לגרום

<sup>87</sup> <https://www.geeksforgeeks.org/two-level-paging-and-multi-level-paging-in-os/>

<sup>88</sup> מתחיל אחרי החזרה מההפסקה, 1:18:40.

<sup>89</sup> <https://www.geeksforgeeks.org/inverted-page-table-in-operating-system/>

לכך שהרבה מאוד משאבים הולכים לניהול הטבלאות ולא לתהליכים שרצים בה.

- ❑ Regular page tables impractical for 64-bit address space  
4K page size /  $2^{52}$  pages x 8 bytes → 30M GB page tables!
  - ❑ Inverted page table – sorted by (physical) page frames and not by virtual pages  
1 GB of RAM & 4K page size / 256K entries → 2 MB table
  - ❑ A single inverted page table used for all processes currently in memory
  - ❑ Each entry stores which process/virtual-page maps to it
  - ❑ A hash table is used to avoid linear search for every virtual page
  - ❑ In addition to the hash table, TLB registers are used to store recently used page table entries
- 

b. הפתרון (בתמונה הנ"ל):

- i. נייצר טבלה שהיא הפוכה, ממופה ע"י כתובות פיזיות ולא וירטואליות<sup>90</sup>.
- ii. טבלה אחת כזאת תשמש לכל התהליכים שכרגע רצים, כל רשומה בטבלה שומרת את העמוד הווירטואלי או התהליך שממופה לה.
- iii. נשתמש במבנה נתונים מסוג Hash-Table<sup>91</sup> בכדי לגשת לטבלאות של כל תהליך וזאת ע"י פנייה גם לכתובת וירטואלית וגם ל-PID<sup>92</sup> של אותו תהליך במקביל. במצב כזה, ככל שרצים יותר תהליכים על המערכת, מאוד סביר ששניים יפנו לאותו מקום. כמו כן נשתמש ב-TLB<sup>93</sup>.
- iv. הערה: נשים לב שאנחנו נרצה מנגנון כלשהו ששומר את הכתובות שמצאנו (מנגנון Caching מסוים) בכדי לחסוך זמן בהמשך, וזה בעיקר בעקבות מנגנון ה-Locality of references<sup>94</sup> מאוד נפוץ במדעי המחשב, מנגנון שגורם לתהליכים נוטים לפנות לאותם כתובות זיכרון ושוב ושוב בצורה רפטטיבית.

#### 49. PTE – Page Table Entries:

- a. Page frame number (physical address) – הכתובת הפיזית שניגשים אליה
- b. Present/absent bit (valid or nor) – האם הדף נמצא בזיכרון? כן = 1.
- c. Dirty bit (modified or not) – האם מישחו שינה את הדף? כבר בשימוש? הדבר משפיע על הכתיבה של העמוד לדיסק וכמו כן משפיע על הרבה אלגוריתמים שרצים על התהליכים והזיכרון.
- d. Referenced bit (accessed or not) – בודקים האם בשימוש או לא, מסייע לקראת ניקוי. פעם בכמה זמן אנחנו בודקים איזה זיכרון לזרוק/לפנות.
- e. Protection – זכויות גישה? אולי זה Read Only. הדוגמא זה ה-ROM, או משתני const למיניהם ועוד. כמו כן יש דפים שהם Executable, כגון code segment ועוד.
- f. Caching disable/enable – לפעמים אנחנו נרצה למפות זיכרון בצורה שלא יהיה קאשינג כי זה יכול לפגוע בביצועים. מקרה נוסף שנרצה למנוע זה זיכרון שיש בו רגיש שלא נרצה שימופה לקובץ או Cache כזה או אחר.

50. MMU – Memory Management Unit: כל יחידת זיכרון/רשומת זיכרון שמתרגמת זיכרון פיזי לוירטואלי, בדומה לתהליך ה-Paging שתואר מעלה.

#### 51. TLB – Translation Lookaside Buffer:

- a. הערה: חשוב לזכור שיש Software TLB שקיים בתהליכים ויש גם Hardware TLB שנמצא בתוך ה-MMU של המעבד<sup>95</sup>.
- b. תהליך השאילתה (Resolving): התהליך/קרנל פונה לתרגום של תהליך וירטואלי.
- c. השאילתה מגיעה ל-TLB/MMU, אם הוא קיים אצלו, הוא מחזיר את הכתובת.
- d. אחרת, במידה וה-TLB/MMU לא מצליח להחזיר את הכתובת אבל הכתובת חוקית<sup>96</sup>.

<sup>90</sup> בתמונה המצורפת ניתן לראות את החישוב במקרה של 1 ג'יגה ראם.

<sup>91</sup> נזכור שהחיפוש הוא על כל המערכת הפעלה, כל התהליכים שרצים במקביל ולכן נרצה את הביצועים הטובים ביותר בשליפה. לכן הסיבה שמשתמשים במבנה הזה הוא כדי למנוע חיפוש לינארי שעובר על כל הרשומות, כל נוכל לפנות לערך נתון (לפי ה-PID) הרבה יותר מהר.

<sup>92</sup> נזכור ש PID הוא ערך ייחודי לכל תהליך ועל כן אפשר להשתמש בו לשליפה.

<sup>93</sup> יוסבר בהמשך.

<sup>94</sup> [https://en.wikipedia.org/wiki/Locality\\_of\\_reference](https://en.wikipedia.org/wiki/Locality_of_reference)

<sup>95</sup> <https://www.geeksforgeeks.org/page-table-entries-in-page-table/>

<sup>96</sup> [https://en.wikipedia.org/wiki/Memory\\_management\\_unit](https://en.wikipedia.org/wiki/Memory_management_unit)

<sup>97</sup> מתחיל ב-1:35:00, המרצה ציין שזה נושא מספיק חשוב אפילו להרצאה כולה.

<sup>98</sup> כמו כן חשוב לזכור שישנם מעבדים ללא MMU כלל, במקרה כזה יש רק SOFTWARE, זה אמנם מקרה נדיר, אבל קיים.

<sup>99</sup> כלומר הכתובת ממופה, קיימת בטבלה / הוקצה לה טבלה.



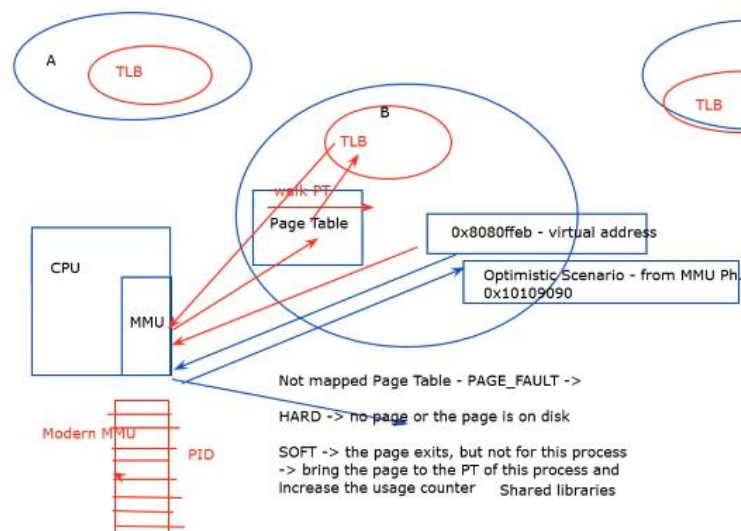
- i. במקרה והכתובת חוקית, קיימת ב-TLB, אזי מטיילים ב-PT ומעדכנים את ה-TLB בהתאם.
- ii. במקרה והכתובת לא חוקית/לא ממופה<sup>100</sup>, במקרה כזה נקבל שגיאת **PAGE\_FAULT**<sup>101</sup>. קיימים שני סוגים של השגיאה הנ"ל וטיפול בהם<sup>102</sup>:
  - (1) **SOFT**<sup>103</sup> – כאן קיים הדף אבל הוא מוקצה לתהליך אחר, כלומר, עושים אינקרמנטציה למספר השימושים.
  - (2) **HARD** – כאן מקצים דף חדש/זיכרון פיזי חדש כי הוא בכלל לא קיים בזיכרון.
  - (3) בשני המקרים, אנחנו נמפה את העמוד ונכניס ל-PT.
- iii. במקרה שהכתובת ממופה, אבל ב-MMU אין מיפוי: נפנה ל-Page Table ונעשה מה שנקרא **Walk**, נסרוק את הטבלה ונראה את הרישומים, אחרי שנאתר את הכתובת אנחנו נשמור אותה ב-TLB.

## 52. שאלות חשיבה ותרגול<sup>104</sup>:

- a. שאלה: למה מנגנון ה-Two-Level Page table חוסך זיכרון?
- b. שאלה: למה Page Table רגיל לא מתאים למערכת 64 ביט?
- c. שאלה<sup>105</sup>: מה ההבדל בין Context Switch של חוטים של אותו תהליך לבין תהליכים שונים?  
תשובה: ניתן להבין מהשאלה ש-TLB מומש עם Entry אחד לכן כאשר עושים Context Switch בחוטים של תהליכים שונים במצב הזה מביאים למצב של **TLB-Flush**<sup>106</sup> והעמסה של TLB יחדשים, זה למה במקרה הזה יקח לנו יותר זמן עיבוד.
- d. שאלה: אותה שאלה רק במעבדים חדשים יותר.  
תשובה: ניתן לראות שיש TLB עם רשומות מרובות ועל כן FLUSH לא רלוונטי.

## הרצאה 12:

## 53. חזרה על הרצאה 11<sup>107</sup>:



בתמונה הנ"ל ניתן לראות מספר תהליכים והדרך שהם מתנהלים למול ה-TLB/MMU, בין אם שלהם ובין אם של החומרה. סורקים מחדש את התהליך מהחדש שתהליך פונה ל-MMU לקבלת כתובת פיזית, מה קורה כאשר הוא לא מצליח ומה השגיאות האפשריות<sup>108</sup>. החידוש בשיעור בשונה מהשיעור הקודם שכאן שמנו דגש על תהליך ה-Walk שכמעט ולא הוזכר קודם. במהלך ה"הליכה" נבצע סריקה על הכתובות בטבלה שלנו לראות איפה הכתובת שאנחנו מחפשים ואותה נשמור ב-TLB לטובת שיפור ביצועים.

<sup>100</sup> כלומר עדיין לא הקצו למקום הזה דף פיזי.

<sup>101</sup> [https://en.wikipedia.org/wiki/Page\\_fault#:~:text=A%20page%20fault%20\(sometimes%20called,address%20space%20of%20a%20process](https://en.wikipedia.org/wiki/Page_fault#:~:text=A%20page%20fault%20(sometimes%20called,address%20space%20of%20a%20process)

<sup>102</sup> במקרה כזה יש שגיאה ב-CPU וצריך למפות את הדף הזה.

<sup>103</sup> בדרך כלל נובע כתוצאה מ-shared libraries, מכיוון שהספריות משותפות, לשני תהליכים יש את אותו דף, ללא הקצאה של דף פיזי, עושים את זה כדי לחסוך בזיכרון. נחשוב על מצב שיש לנו הרבה מאוד תהליכים שמשתמשים באותו דף, זה הגיוני מאוד כי לא צריך הקצאת זיכרון לכל אחד על ההתחלה.

<sup>104</sup> חלק מהשאלות לא בהכרח היו במהלך השיעור, אלא נרשמו בעקבות סוגיות בהן המרצה התרכז.

<sup>105</sup> שאלת ראיון עבודה, 1:37:30

<sup>106</sup> כלומר, התהליך החדש מחליף את הרשומות ב-MMU לרשומות שלו.

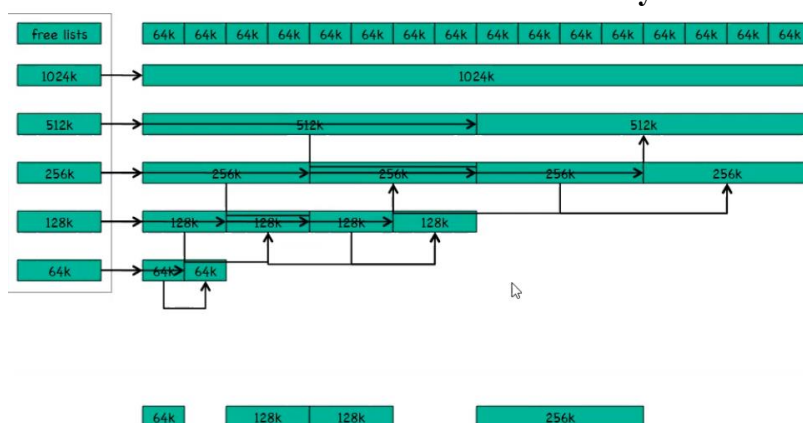
<sup>107</sup> מתחילת השיעור עד 27:00

<sup>108</sup> פירוט מלא נמצא בסיכום של הרצאה 11



54. **כיצד עובד Dynamic Allocation** : כאשר אנחנו מריצים calloc/malloc לטובת ההקצאה הדינאמית הקוד פונה ב-syscall לספריה <sup>109</sup>**GLIBC - Library Allocator**, אפ'י ש לו זיכרון הוא מקצה לנו אחת הוא פונה ל-Kernel, גם כאן הזיכרון הוא ביחידות דף. הלוגיקה כאן נקראת Knuth's Buddy Allocator.

#### 55. <sup>110</sup>**Knuth's Buddy Allocator**



מנגנון שמומש היום כמעט בכל Kernel ומערכות הפעלה.

#### 56. נושאים שלא הספקנו ללמוד בקורס<sup>111</sup> :

1. Synchronization: Tournament Tree and Lamport's Bakery Algorithms.
2. Synchronization: Monitors and barriers. Monitors in Java. Event counters and messages.
3. Synchronization: The Readers and Writers Problem. Implementation of the Read-Write Lock by binary semaphores preventing starvations of the readers and the writers.
4. Synchronization: Sleeping Barber Problem.
5. Synchronization: The Mellor-Crummey and Scott (MCS) Multi-Core Friendly Algorithm.
6. Memory Management: Multi-Level and inverted page tables in-depth.
7. Memory Management: page replacement algorithms: FIFO, second chance FIFO, LRU, NFU, the clock algorithm, working set and WS clock. Implementation issues in paging.
8. Memory management: segmentation, memory management in user mode, heap manager and memory mapped files, shared memory, memory locking and segmentation in the Pentium architecture
9. File systems: directories and file types, file management, file system implementation, FAT, UNIX file system, MS-DOS file system, disk management, file system reliability, NTFS and the basics of the SSD/Flash FS.
10. Virtualization and cloud computing: hypervisors, virtual machines, KVM and Open stack
11. I/O in OS: interrupts, I/O ports, memory-mapping and DMA.

#### 57. שאלות חשיבה ותרגול :

a. **שאלה** : למה צריך TLB לכל תהליך?

**תשובה** :

b. **שאלה** : למה כל הכתובות קאשינג של כתובות וירטואליות מתחזקות פעמיים?

**תשובה** : זה קורה פעם ב MMU ופעם ב TLB. זאת מכיוון שברגע שנעוף מה-CPU יכול לבוא תהליך אחר ולעשות FLUSH ואנחנו נצטרך לטעון מחדש<sup>112</sup>.

<sup>109</sup> <https://www.gnu.org/software/libc/>

<sup>110</sup> מתחיל 32:00

<sup>111</sup> עלה לקראת סוף השיעור האחרון, המסמך נמצא במודל, מתחיל ב 38:00

<sup>112</sup> צריך להתייחס פה גם למקרה שיש מעבדים חדשים עם MMU מודרני שם יש לו כמה כניסות. שם תהליך יכנס עם PID שזה ערך יחודי ועל כן הוא לא יהיה חייב לעשות FLUSH. זה גורם לכך שה CS יהיה מהיר יותר.