מערכות הפעלה - הרצאה 10

שרון מלטר, אתגר 17

28 ביולי 2024

1 *Chapter 7 Cont.*

נעבור לבעיה החדשה. פילוסופים.

1.1 בעיית הפילוסופים הסועדים

יהיו n פילוסופים. הם מבלים את כל חייהם באכילה וחשיבה מסביב לשולחן (ועוד נותנים מלגות למדעי הרוח) כאשר פילוסוף לא חושב, הוא רעב, ואז הוא צריך שני צ'ופסטיקים בשביל לאכל. ישנו צ'ופסטיק אחד בין כל זוג פילוסופים, ורק אחד מהם יכול לקחת אותו בכל פעם.
נפתור את הבעיה בעזרת סמפורים, להלן מימוש של פילוסוף; בעיה: אם פילוסוף i לוקח שני צ'ופסטיקים, אבל

```
while (true){
    wait (chopstick[i] );
    wait (chopStick[ (i + 1) % 5] );

     /* eat for awhile */

    signal (chopstick[i] );
    signal (chopstick[ (i + 1) % 5] );

     /* think for awhile */

}
```

מחזיר לפני ש- $i-1$ או $i+1$ רעבים, אז כאשר הם יהיו רעבים, הצ'ופסטיקס לעולם לא ישתחררו עבורם...

ניסיון שני!
נפתור את הבעיה עם מוניטור, עבור $n = 5$; הסבר:

```
monitor DiningPhilosophers
{
   enum {THINKING; HUNGRY, EATING} state [5];
   condition self [5];

   void pickup (int i) {
          state[i] = HUNGRY;
          test(i);
          if (state[i] != EATING) self[i].wait;
   }

   void putdown (int i) {
          state[i] = THINKING;
                   // test left and right neighbors
          test((i + 4) % 5);
          test((i + 1) % 5);
   }


   void test (int i) {
           if ((state[(i + 4) % 5] != EATING) &&
           (state[i] == HUNGRY) &&
           (state[(i + 1) % 5] != EATING) ) {
                state[i] = EATING ;
               self[i].signal () ;
           }
   }

       initialization_code() {
          for (int i = 0; i < 5; i++)
          state[i] = THINKING;
        }
}
```

1. $pickup(i)$ - הפילוסוף i רוצה לאכל. המצב שלו משתנה ל- $HUNGRY$ והוא בודק עם $test(i)$ האם הוא כעת יכול לקחת את שני הצ'ופסטיקים שמצדידו. אם לא, הוא מחכה עם $self[i].wait()$.
2. $putdown(i)$ - הפילוסוף עובר למצב $EATING$ ובודק האם אחד משכניו יכול לאכל כעת.
3. $test(i)$ - פעולה הבודקת האם פילוסוף יכול ורוצה לאכל לאכל, ואם כן מעבירה אותו ל- $EATING$ וקוראת ל-- $self[i].signal$ כדי שייצא מ- $wait()$
4. $initialization\_code()$ - פעולה המאתחלת את מצבי כל הפילוסיפים ל- $THINKING$.

ההבדל העיקרי הוא שכעת מוודאים כי פילוסוף יוכל לאכל ברגע שבו הוא רעב או ברגע שבו כל שכניו מפסיקים לאכל.

אזי אין יותר בעיית $deadlock$, אך כן תיתכן הרעבה.
מכאן רק דוגמאות ממערכות הפעלה... עד פה פרק 7 :)

2 Chapter 8 - Deadlocks

2.1 Outline

- System Model
- Deadlock Characterization
- Methods for Handling Deadlocks
- Deadlock Prevention
- Deadlock Avoidance
- Deadlock Detection
- Recovery from Deadlock

2.2 Objectives

- Illustrate how deadlock can occur when mutex locks are used
- Define the four necessary conditions that characterize deadlock
- Identify a deadlock situation in a resource allocation graph
- Evaluate the four different approaches for preventing deadlocks
- Apply the banker's algorithm for deadlock avoidance
- Apply the deadlock detection algorithm
- Evaluate approaches for recovering from deadlock

2.3 מודל מערכת

נתאר את מודל המערכת על מנת להגדיר את הבעיה איתה נתמודד בפרק זה.
נניח שסוגי המשאבים של מערכת כלשהי הם $R_1, R_2, ..., R_m$ (מעגלי CPU, מרחב זיכרון, התקני I/O) כך שלמשאב R_i יש W_i מופעים. כמו כן, כל תהליך יכול לנצל משאב בעזרת שלושת הפעולות הבאות:

- $request$
- use
- $release$

כעת נעבור לבעיה עצמה.

2.4 דדלוק עם סמפורים

נניח שיש לנו שני סמפורים המאותחלים ל־ 1, S_1 ו־ S_2 וכמו כן שני חוטים T_2 T_1,.

- T_1 :
  1. $wait(S_1)$
  2. $wait(S_2)$
- T_2 :
  1. $wait(S_2)$
  2. $wait(S_1)$

כעת גם S_1 וגם S_2 מחכים לסיגנל, אבל שני החוטים המשתמשים בהם, T_1 ו־ T_2 לא יכולים לרוץ כך שהוא לעולם לא ייקרא.
זהו מקרה פרטי. נדבר על ארבעת התנאים שצריכים להתקיים על מנת לגרום לדדלוק.

2.5 התנאים לדדלוק

דדלוק מתרחש כאשר ארבעה תנאים אלה מתקיימים במקביל;

- $Mutual\ Exclusion$ ־ רק חוט אחד בכל פעם יכול להשתמש במשאב.
- $Hold\ and\ wait$ ־ חוט אשר מחזיק במשאב אחד לפחות מחכה למשאב נוסף שמוחזק אצל חוט אחר.
- $No\ Preemption$ ־ משאב יכול להשתחרר מהחוט המחזיק בו רק כאשר החוט סיים את משימתו.
- $Circular\ Wait$ ־ קיים סט $T_0, T_1, ..., T_n$ של חוטים המחכים למשאב כך ש־ T_0 מחכה למשאב שמחזיק T_1, T_1 למשאב שמחזיק T_2 וכן הלאה. כמו כן, T_n מחכה למשאב שמחזיק T_0.

הפרה של כל אחד מהתנאים גורמת לכך שלא ייתכן דדלוק.
איך פותרים / מונעים דדלוקים? ־ תחילה נסתכל עליהם בדרך אחרת וקצת יותר ויזואלית.

2.6 גרף הקצאת משאבים

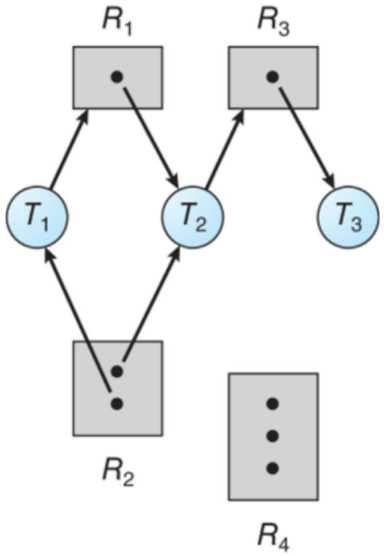
יהי סט צמתים V וסט קשתות E. V מחולק לשני סוגי צמתים:

- $T = \{T_1, T_2, ..., T_n\}$ סט שבו כל חוטי המערכת.
- $R = \{R_1, R_2, ..., R_m\}$ סט של כל סוגי המשאבים במערכת.

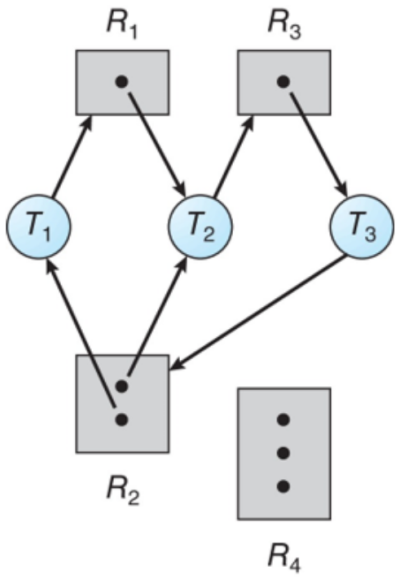
כמו כן, כל קשת ב־ E היא אחת משני סוגים;

- $request\ edge$ ־ קשת מכוונת $T_i \rightarrow R_j$
- $assignment\ edge$ ־ קשת מכוונת $R_j \rightarrow T_i$

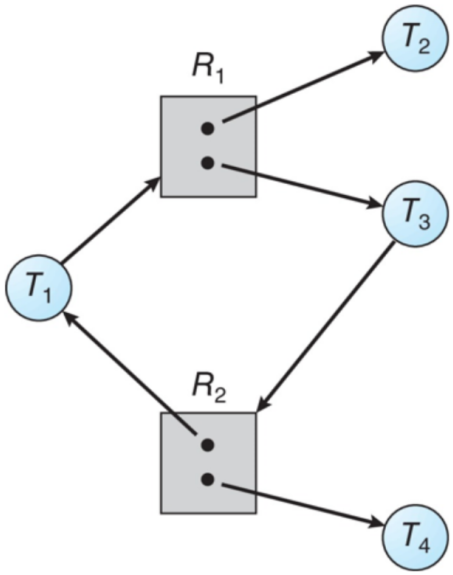
כלומר, V, E מהווים גרף המייצג את הקצאות ובקשות ההקצאה של חוטי ומשאבי המערכת. להלן דוגמה;



אבל זו הייתה דוגמה מעט משעממת. מה לגבי דדלוק?



שימו לב ♡
לא כל גרף הקצאת משאבים שיש בו מעגל הוא עם דדלוק. זהו תנאי הכרחי אך לא מספק. למשל; במקרה זה



חוט T_4 לא מחכה למשאב אחר ולכן יכול לשחרר מופע של R_2 ובכך לאפשר את המשך משימות שאר החוטים.
באופן כללי, לפי התנאים שלמדנו, על מנת שיהיה דדלוק נדרש להיות מעגל בגרף, אך אם יש למשאבים שבו מספר מופעים ייתכן שאין דדלוק.
התעסקנו בלדבר על הבעיה יותר מדי. הגיע הזמן לעבור לשיטות פתרון.

2.7 שיטות לפתרון דדלוקים

ישנן שלוש שיטות עיקריות:

- ווידוא שהמערכת לעולם לא תיכנס למצב דדלוק, בעזרת מניעת או הימנעות מדדלוקים.
- לתת למערכת להיכנס לדדלוק אבל לאפשר להוציא אותה ממנו.
- להתעלם ולהעמיד פנים שדדלוקים לעולם לא יקרו במערכת (כזכור ניתן עם $timer$ לוודא שנצא מלופ אינסופי)

2.7.1 *Deadlock Prevention*

בשיטה זו מבטלים את אחד מארבעת התנאים הנדרשים ליצירת דדלוק. להלן כיצד ניתן ליישם זאת עבור כל אחד מארבעת התנאים:

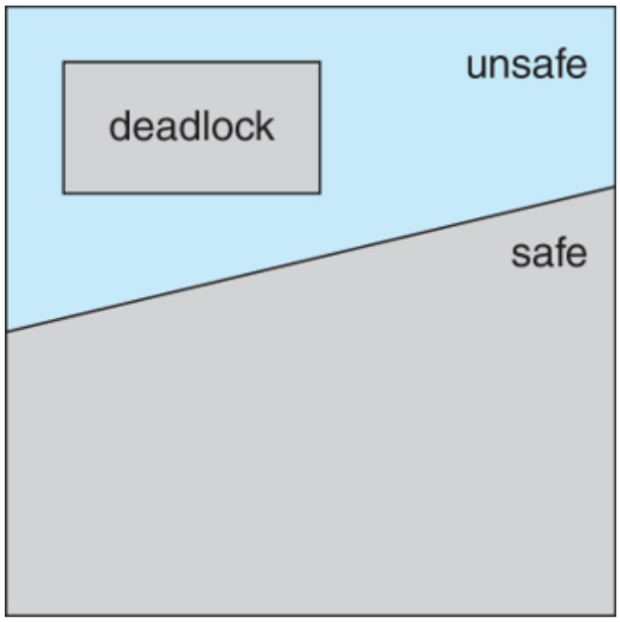
- *Mutual Exclusion* - ניתן לוותר על אלמנט זה, למשל עבור קבצים משותפים שניתן רק לקרוא מהם (כזכור מבעיית קוראים־כותבים)
- *Hold and Wait* - ווידוא כי כאשר תהליך מבקש משאב, הוא לא מחזיק אף משאב אחר. בפתרון זה ישנו ניצול מועט של המשאבים, כלומר תיתכן הרעבה של משאב.
- *No Preemption* - אם תהליך מחזיק במשאבים ומבקש משאב אחר, הוא משחרר מיד את שאר המשאבים. כלומר, עושים *preemption* למשאבים כך שאותו תהליך דורש אותם מחדש. התהליך ימשיך במשימתו רק לאחר שהשיג אותם מחדש וגם את המשאב הנוסף שביקש.
- *Circular Wait* - ניצור סדרה של כל סוגי המשאבים ונוודא כי כל תהליך מבקש משאבים מסדר עולה.

האפשרות הנפוצה ביותר מבין הרשימה היא ביטול *circular wait* (האפשרות האחרונה)
בסך הכל נדרש להקצות לכל משאב מספר ייחודי ולוודא כי התהליכים מבקשים אותם בסדר עולה.

2.7.2 *Deadlock Avoidance*

פתרון זה דורש שלמערכת יהיה מידע התחלתי זמין.
במודל הכי פשוט, יש רק לדעת את מספר המשאבים המקסימלי מכל סוג שחוט יכול להזדקק לו. אלגוריתם ההימנעות מדדלוקים בודק את מצב ההקצאות הנוכחי על מנת לוודא שלא יכולה להיווצר סיטואציה של *circular wait*.
מצב הקצאות המשאבים מוגדר על ידי מספר המשאבים החופשיים והמוקצים והדרישות המקיסמליות של התהליכים.

מצב בו אין סכנת *circular wait* מוגדר *Safe State*.
איך מזהים מצב בטוח? זהו מצב בו קיימת סדרה $< T_1, T_2, ..., T_n >$ של כל חוטי המערכת כך שלכל T_i המשאבים שהוא עדיין יכול לבקש הם חופשיים או מוקצאים רק לחוט T_j כך ש־ $j > i$. כך T_i יכול לקחת אותם במיידיות או רק לחכות לכך שהתהליכים שאחריו יסיימו, ללא חשש מכך שהם יצטרכו לחכות לו.
כאשר חוט מבקש משאב, המערכת חייבת להחליט האם ההקצאה משאירה אותה במצב בטוח.
להלן תרשים מתוחכם המדגים את מצבי המערכת האפשריים; הערה: מצב בו קיים דדלוק מכונה *deadlock state*.



אז מוודאים שאנחנו תמיד במצב בטוח?

קיימים שני אלגוריתמים. כאשר קיים רק מופע אחד של כל משאב, משתמשים בגרף הקצאות משאבים. למה? כי כזכור, כאשר יש מעגל בגרף ואין מספר מופעים למשאבים שבו, בהכרח יש דדלוק.
בשאר המקרים משתמשים באלגוריתם הבנקאי, $Banker's\ Algorithm$.

עבור האלגוריתם הראשון יש רק לבנות גרף מהחוטים והמשאבים ולבדוק באמצעות אלגוריתם סריקה האם קיים בו מעגל עבור הקצאה תיאורטית.
נלמד מהו אלגוריתם הבנקאי.

בשביל האלגוריתם נצטרך לדעת מהי הדרישה המקסימלית של כל חוט לסוגי המשאבים. לאחר שכל אחד מהם מוקצה אליו, הוא חייב לשחרר אותם לאחר זמן סופי.
ישנם מספר מבני נתונים שהאלגוריתם דורש;

- $Available$ - וקטור באורך m הסופר את מספר המופעים החופשיים של כל סוג משאב. למשל אם $available[i] = k$ אזי ישנם k מופעים פנויים של המשאב R_i.
- Max - מטריצה מסדר $n \times m$ הסופרת את מספר המופעים המקסימלי שכל חוט יכול לדרוש מכל סוג משאב.
- $Allocation$ - מטריצה מסדר $n \times m$ שסופרת כמה מופעים של כל סוג משאב מוקצה לכל חוט.
- $Need$ - מטריצה מסדר $n \times m$ הסופרת מהו כמות הדרישות הנוספות המקסימלי שכל חוט יכול לבקש מכל סוג של משאב, כלומר: $Need[i,j] = Max[i,j] - Allocation[i,j]$

כעת נוכל לעבור לאלגוריתם הביטחון:

1. יהיו $Work$ ו- $Finish$ שני וקטורים מאורכים m ו- n בהתאמה, כך ש-

$$Work = Available$$

(כן, זה מסבך שלא לצורך...)

$$Finish[i] = false \quad for \quad i = 0, 1, .., n-1$$

2. נמצא i כך ש- $Finish[i] = false$ וגם $Need_i \leqslant Work$. אם לא קיים i כזה, נקפוץ לשלב 4.
3.

$$Work = Work + Allocation_i$$

$$Finish[i] = true$$

לחזור לשלב 2.

4. אם $Finish[i] == true$ לכל i, אז המערכת במצב בטוח.

וכך בודקים אם המערכת נמצאת או תהיה במצב בטוח :)
אך עדיין יש פה קופסא שחורה, איך מקצים את המשאבים?

יהי הווקטור $Request_i$ וקטור של בקשות ההקצאה של התהליך T_i כך שאם $Request_i[j] = k$ אזי T_i מבקש k מופעים של סוג המשאב R_j.
להלן אלגוריתם הקצאת המשאבים אשר ישמש אותנו:

1. אם $Request_k \leqslant Need_i$, עבור לשלב 2. אחרת, קרא לשגיאה.
2. אם $Request_i \leqslant Available$, עבור לשלב 3. אחרת, T_i חייב לחכות לפינוי של משאבים נוספים.
3. נעמיד פנים שאנו מקצים ל- T_i את המשאבים כך;

$$Available = Available - Request_i$$

$$Allocation_i = Allocation_i + Request_i$$

$$Need_i = Need_i - Request_i$$

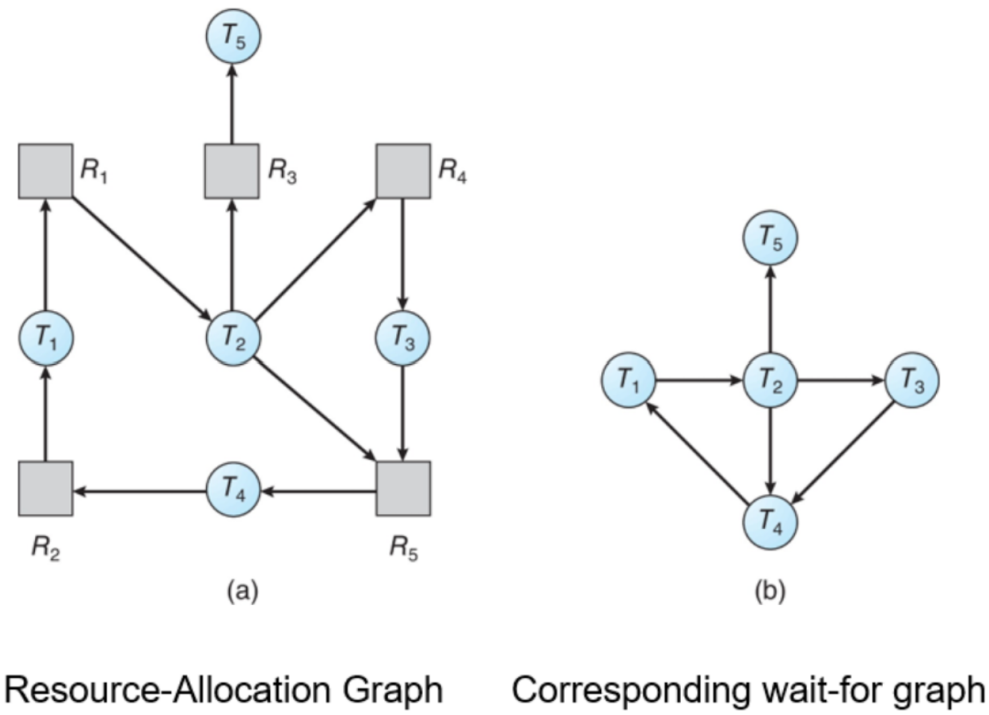
אם הגענו למצב בטוח, ניתן להקצות ל־ T_i את המשאבים. אחרת, T_i יצטרך לחכות לכך שמשאבים נוספים יפונו.

עד כאן פתרונות מניעתיים לדדלוק. נעבור לשיטה הבאה.

2.8 זיהוי דדלוק

בשיטה זו נאפשר לדדלוקים לקרות, אך נזהה ונפתור אותם.

נתחיל מהמקרה של מופע יחיד לכל משאב.
נתחזק גרף $wait-for$, בו כל צומת הוא חוט ואם קיימת קשת $T_i \to T_j$ אזי החוט T_i מחכה לחוט T_j.
פעם בכמה זמן, נבדוק האם קיים בגרף זה מעגל (באמצעות אלגוריתם סריקה). אם קיים מעגל, קיים דדלוק.
להלן דוגמה לגרף $wait-for$; פשוט וחביב :)



כעת למקרה שבו ישנם מספר מופעים אפשריים למשאב.
ניעזר שוב במבני הנתונים $Available$, $Allocation$, $Request$
וכך נבנה את אלגוריתם הזיהוי:

1. יהיו $Work$ ו־ $Finish$ שני וקטורים מאורך m ו־ n בהתאמה. נאתחל $Work = Available$ ולכל $i = 1, 2, ..., n$, אם $Allocation_i \neq 0$ אזי $Finish[i] = false$ ואחרת $Finish[i] = true$.
2. נמצא i כך ש־
   - $Finish[i] == false$
   - $Request_i \leqslant Work$

אם לא קיים i כזה, עבור לשלב 4.

3. $Work = Work + Allocation_i$
$Finish[i] = true$
ונעבור לשלב 2.

4. אם $Finish[i]$ עבור $1 \leqslant i \leqslant$ כלשהו, אז המערכת נמצאת במצב דדלוק. בפרט, החוט T_i נמצא בדדלוק.

בסך הכל סיבוכיות זמן $O(m \times n^2)$ כדי לבדוק האם המערכת במצב דדלוק. מדובר באלגוריתם יקר, כך שקצב הקריאות אליו תלוי בסבירות שיתרחש דדלוק וכמה חוטים בד"כ יצטרכו לשחרר משאבים ולחכות (כלומר כמה מעגלים בלתי תלויים מתקבלים) אם נריץ את אלגוריתם הזיהוי באופן שרירותי, לא ניתן יהיה לדעת איזה מהחוטים "גרם" לדדלוק.
אבל מה בתכל'ס עושים כשמוצאים דדלוק? - ישנן שתי שיטות להיפטר ממנו.

2.8.1 ביטול תהליך - $Process\ Termination$

הגישה הראשונה היא לבטל את כל החוטים שבדדלוק (שלא סיימו את משימתם). כמובן שכך הדדלוק נפתר.

גישה מעט יותר עדינה היא לבטל תהליך בכל אחד כל פעם עד שאין מעגל. להלן מספר פרמטרים שניתן להיעזר בהם כדי לבחור חוט שיבוטל;

- עדיפות החוט.
- כמה זמן החוט רץ וכמה זמן עוד נותר לסיום משימתו.
- משאבים בהם החוט משתמש.
- המשאבים שהחוט זקוק בנוסף למה שהוקצה לו.
- כמה חוטים יצטרכו להיבטל.

2.8.2 $Preemption$

בשיטה זו נבחר קורבן ונשחרר את ההקצאות שלו.
ישנה סכנה של הרעבה במקרה בו אותו החוט תמיד נבחר כקורבן.

עד כאן פרק 8. ובפרקים הבאים- ניהול זיכרון.

3 Chapter 9 - Main Memory

3.1 Outline

- Background
- Contiguous Memory Allocation
- Paging
- Structure of the Page Table
- Swapping
- Example: The Intel 32 and 64-bit Architectures
- Example: ARMv8 Architecture

3.2 Objectives

- To provide a detailed description of various ways of organizing memory hardware
- To discuss various memory-management techniques,
- To provide a detailed description of the Intel Pentium, which supports both pure segmentation and segmentation with paging

נתחיל מסיפור הרקע הטראגי שלנו.

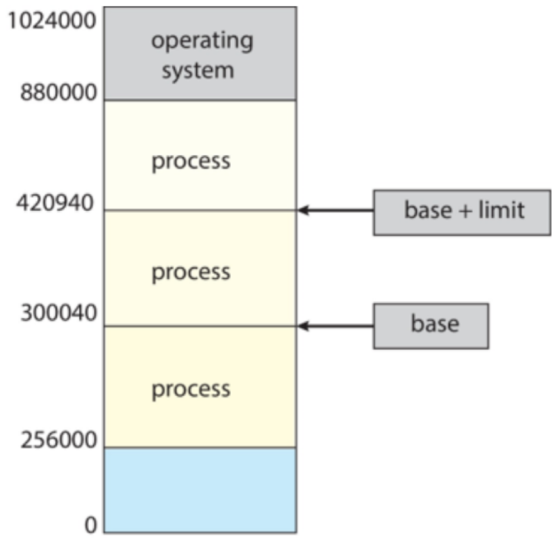
3.3 Background

תוכנה חייבת לעבור מהדיסק לזיכרון הראשי ולהתחבר לתהליך על מנת לרוץ. זאת מכיוון שהזיכרון הראשי והרגיסטרים הם האחסון היחיד שה־ CPU יכול לגשת אליו באופן ישיר. עם זאת, הם לא שווים־ זמן הגישה לרגיסטר הוא לכל היותר $CPU\ cycle$, אך גישה לזיכרון הראשי יכולה לקחת מספר מעגלים, מה שיגרום לעיכוב. הפתרון הוא $Cache$ שיושב בין הזיכרון הראשי לרגיסטרים של ה־ CPU. הערה: כל יחידת זיכרון היא כתובת ובקשת קריאה או כתובת ובקשת דאטה וכתיבה. יש לוודא כי נתוני הזיכרון לא משתנים בין ה־ $Cache$ לזיכרון הראשי.
נעבור לדבר על המנגנונים שמסייעים לנו לנהל את הזיכרון הראשי.

3.4 הגנה - Protection

יש לוודא כי כל תהליך יכול לגשת רק לכתובות שבמרחב הכתובות שלו. ניתן להבטיח הגנה זו בעזרת זוג רגיסטרים, $base$ ו־ $limit$, אשר מגדירים את מרחב הכתובות של תהליך.

להלן תרשים המראה את מרחב הכתובות של תהליך בתוך מרחב הזכירון של המערכת;



אבל איך מוודאים שהגישה לכתובת מכל תהליך היא רק למרחב הכתובות שלו? ־ החומרה נחלצת לעזרתנו.
ה־ CPU בודק כל גישה לזיכרון שנוצרת על ידי היוזר כדי לוודא שהיא ממרחב הזיכרון המתאים.
שימו ♡ ־ הפקודות להעלאת הרגיסטרים $base$ ו־ $limit$ הן פריבילגיות.

איך התהליך עצמו מתייחס לכתובות? האם הוא תמיד יודע מהי הכתובת האמיתית?
לא! ־ מכיוון שמשתמשים ב־ $Address\ Binding$

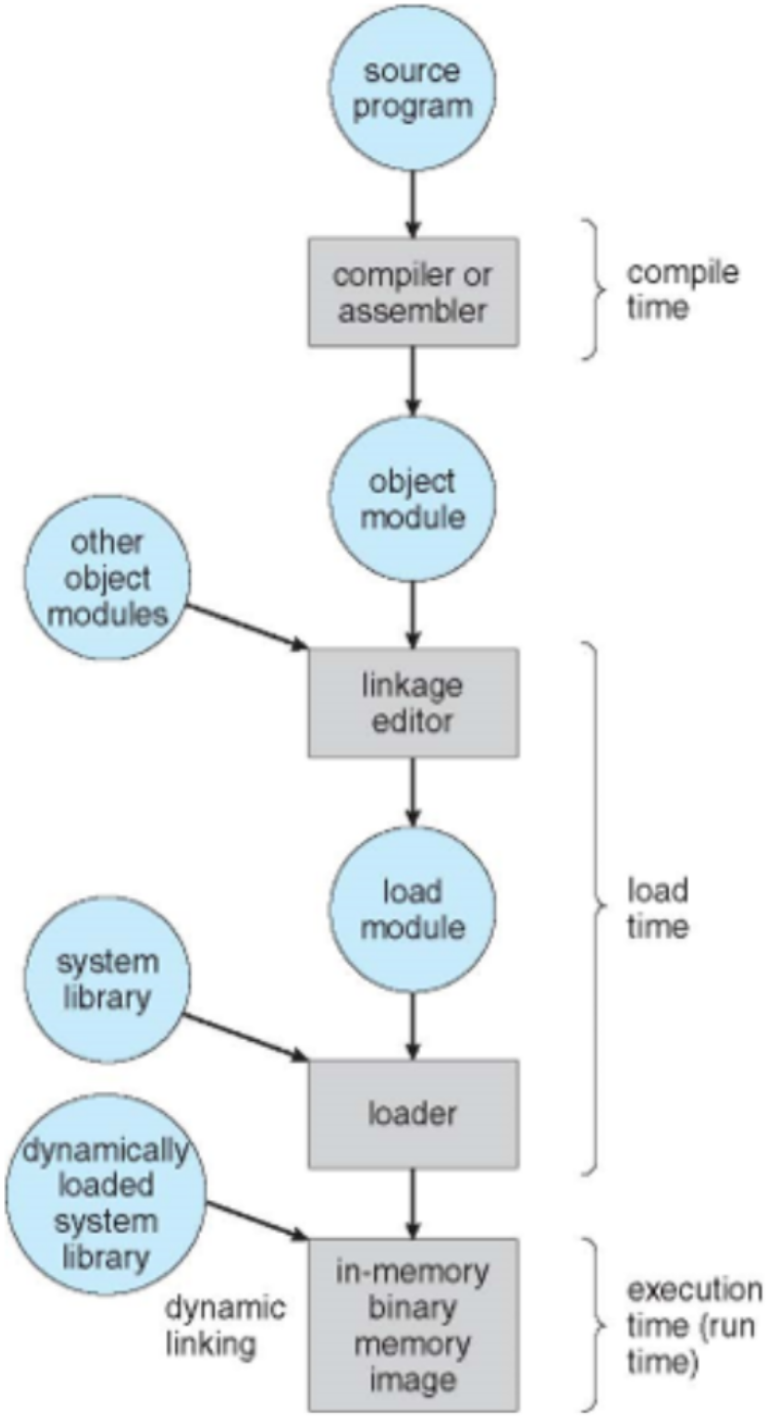
תכניות מחכות להעלאה לזיכרון הראשי מ־ $input\ queue$. ללא תמיכה, יש להעלות אותן לכתובת 0000.
כלומר, באופן אוטומטי זו תהיה הכתובת התחתונה במרחב הכתובות של תהליך, אך אז הכתובת של כל תהליך תצטרך להיות תמיד 0000. מדובר בכתובת פיזית, כך שהיא יחידה.
לכן כתובות של תהליך מיוצגות בדרכים אחרות בשלבים שונים של חיי התכנית.

1. בדרך כלל הכתובות שב־ $source\ code$ הן סמליות.
2. הקוד המועבד עוברים $address\ binding$ לכתובות שניתן לשנות. למשל, "14 בייטים מתחילת מרחב הכתובות"
3. הלינקר או ה־ $loader$ יחברו בין הכתובות הניתנות לשינוי לכתובות מוחלטות, למשל 74014.

יש לבצע $binding$ לכל רכיב בקוד עם כתובת, כלומר לפקודות ולדאטה, יש לבצע $address\ binding$. ניתן לבצע זאת באחד משלושה שלבים.

1. בזמן קומפילציה: אם מיקום תחילת מרחב הכתובות ידוע מראש. יש לקמפל את הקוד מחדש אם הוא השתנה.
2. בזמן העלאה: במקרה זה יש ליצור $relocatable\ code$ אם מיקום הזיכרון לא ידוע בזמן הקומפילציה.
3. בזמן ריצה: ה־ $binding$ נדחה עד להרצה אם התהליך יכול לזוז במהלך הריצה מסגמנט זיכרון אחד לאחר.
   כמו כן במקרה זה נדרשת תמיכה חומרתית מהרגיסטרים $base$ ו־ $limit$.

להלן שלבי העיבוד של תכנית יוזר:



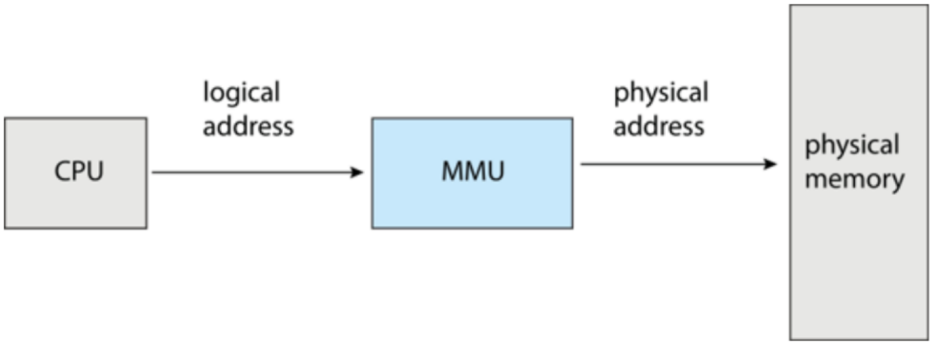
עד כאן הכל טוב ויפה.
עכשיו, האם יש רק סוג אחד של כתובת?
לא - ישנן כתובות לוגיות ופיזיות.

הקונספט של מרחב כתובות לוגי (מכונה גם 'וירטואלי'), הקשור למרחב כתובות פיזי נפרד, הוא בסיס מרכזי בניהול זיכרון. כתובת לוגית נוצרת ע"י ה- CPU וכתובת פיזית היא הכתובת שיחידת הזיכרון רואה.
כתובות לוגיות ופיזיות של אותם האלמנטים הן זהות בזמן הקומפילציה ובזמן ההעלאה, אך שונות בזמן הריצה.
באופן מפתיע מרחב כתובות לוגי מכיל כתובות לוגיות ומרחב כתובות פיזי מכיל כתובות לוגיות.

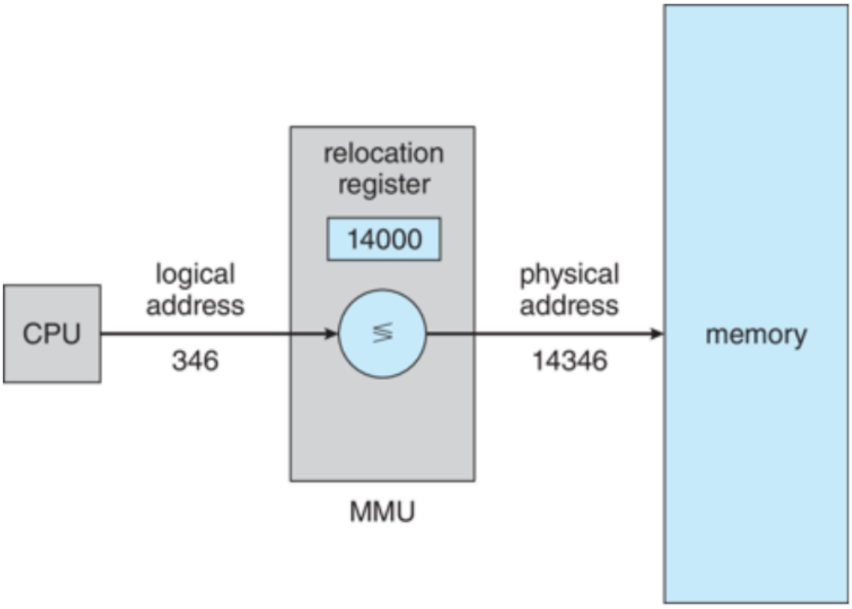
עד כאן, הכל טוב ויפה. ולמסתורין הבא שלנו: מהי יחידת ניהול זיכרון?

3.5 $Memory - Management\ Unit\ (MMU)$

יחידת ניהול זיכרון היא מכשיר חומרה שבזמן הרצה ממפה כתובת וירטואלית לכתובת פיזית. להלן תרשים מתוחכם של היחידה; בשאר הפרק נעבור על הדרכים לבצע מיפוי זה.



עד כה הרגיסטר $base$ הוא שקבע כיצד ממופות הכתובות הווירטואליות, מעתה נקרא לו $relocation\ register$.
הערך שברגיסטר זה נוסף לכל כתובת שנוצרת ע"י תהליך היוזר כאשר היא נשלחת לזיכרון (על מנת לגשת לכתובת הפיזית הנכונה) תכנית היוזר תמיד מתנהלת עם כתובות לוגיות, היא מעולם לא רואה את הכתובות האמיתיות (הפיזיות) $binding$ בזמן ריצה נעשה כאשר ישנו רפרנס לכתובת מהזיכרון.
להלן דוגמה; לפי שיטה זו, גם כל התכנית וגם ה־ $routine$ למיפוי הכתובות צריכות להיות בזיכרון (ה־ $routine$



לא מועלית עד שהיא היא נקראית, כך שניצול המשאבים יעיל יותר) עד ההעלאה, כל הרוטינות שמורות בדיסק כך שניתן להעלות אותן לאחר מכן. גישה זו יעילה גם כאשר קוד רב נדרש כדי להתמודד עם מקרים אשר לא מתרחשים באופן נפוץ.
כמו כן, לא נדרש סיוע מהמערכת להעלאת הרוטינה כשיש צורך, זהו מנגנון המסופק ע"י ארכיטקטורת המערכת.
עם זאת, מערכת יכולה לספק ספריות אשר מיישמות העלאה דינמית.
אבל אי אפשר לדבר על העלאה דינמית בלי לדבר גם על לינקוז' דינמי. (כלשונה של רחל)

3.6 *Dynamic Linking*

קודם נבין מהו *static linking*: ספריות מערכת וקוד תכנית משולבים ע"י ה־ *loader* לתמונת התכנית הבינארית, על מנת לאפשר את הרצת התכנית.

בלינקוז' דינמי הלינקינג נדחה עד זמן ההרצה. כמו שדוחים תרגיל בית.
חלק קטן של קוד, *stub*, מוצא את הספרייה הנדרשת שבזיכרון. ה־ *stub* מחליף את עצמו בכתובת של אותה *routine* ומריץ אותה. מערכת ההפעלה בודקת אם ה־ *routine* נמצאת בכתובת הזיכרון של תהליכים ואם לא אז מוסיפה אותה למרחב הזיכרון שלהם.
מערכת זו ידועה גם בתור *shared libraries* והיא שימושית מאוד מכיוון שהיא חוסכת מה־ *loader* לשלב בין כל תכנית לספריות שלה.

והסוגיה הבאה שלנו ־ איך מקצים כתובות זיכרון?

3.7 *Contiguous Allocation*

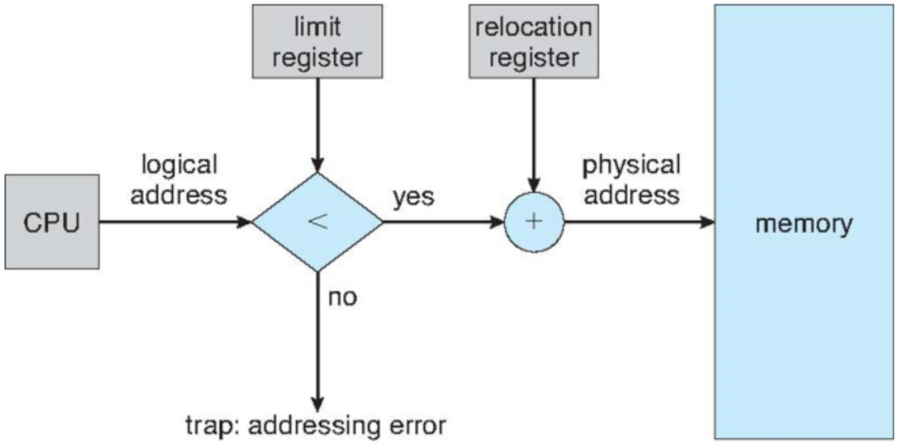
מאחר שתכנית יכולה לרוץ רק כאשר היא בזיכרון הראשי, הוא חייב לתמוך גם בתהליכי יוזר וגם בתהליכי מערכת. כלומר, יש לנו משאבים מוגבלים, כך שנדרש ייעול.
הקצאה רציפה היא שיטת הקצאה ישנה והיא הראשונה שנלמד. בד"כ הזיכרון הראשי מחולק פעמיים;

1. מערכת הפעלה מקומית לרוב מוחזקת בתחתית הזיכרון ביחד עם ה־ *interrupt vector*
2. תהליכי היוזר מוחזקים במקומות גבוהים בזיכרון.
3. כל תהליך מוחזק בחלק יחיד ורציף מהזיכרון.

ידידינו, רגיסטרי ה־ *relocation* מגנים על תהליכי היוזר אחד מהשני ומשינוי הקוד והדאטה של מערכת ההפעלה.
הכיצד? ־

- רגיסטר *base* מכיל את ערך הכתובת הפיזית הקטנה ביותר של התהליך.
- רגיסטר *limit* מכיל טווח של כתובות לוגיות, כך שרגיסטר ה־ *limit* מהווה חסם עליון עבורן.
- ה־ *MMU* ממפה כתובות לוגיות באופן דינמי.
- ישנה אפשרות לבצע פעולות כגון שינוי גודל קרנל והפיכת קוד הקרנל לזמני.

להלן תרשים החומרה התומכת ברגיסטרים *relocation* ו־ *limit*; עוד אלמנט של הקצאה רציפה הוא $Variable-$



partition, כלומר חלוקה של חלקי זיכרון בגודל לא קבוע, בזמן ההרצה־ כאשר ניתן לדעת מהו גודל בלוק הזיכרון שהתהליך צריך
בלוק זיכרון של תהליך נלקח מתוך חור, כלומר אזור רציף ריק בזיכרון.

להלן דוגמה להקצאה רציפה של זיכרון לתהליכים;