היררכיה לפי סדר המהירות: CPU (מהיר מאוד), Cache, זיכרון ראשי, דיסקים, מדפסות, קווי תקשורת, user. נרצה שבין כל רמה ורמה היה מכלא שיקבל אינפורמציה מהרמה הגבוהה, כדי שתוך כדי כך שהרמה הבאה מעבדת את המידע ניתן יהיה להקטין את זמן ההמתנה במעבר מכל שכבה לשכבה. רוצים שלכל היררכיה תהיה <u>אוטונומיה</u> ← אופציה להתחיל לעבוד ולנצל את הרמות הגבוהות יותר.

<u>עקרון השונות – Variance Principle:</u>

צפייה בתוכנית טיפוסית מגלה:

1. תוכניות שונות לא דורשות שירותים זהים באותה עת, ולכן אין הרבה התנגשויות.
2. השרתים השונים לא מועסקים הומוגנית בזמן. זה נותן אפשרות לעבוד ב-*multi programming*.

רעיון השונות מעודד להכניס הרבה תוכניות לאזור הזיכרון כי אפשר ברגעים שונים לספק להם שירותים ומצד שני יש רצון לדמות רכיבים פיזיים ע"י מכלאים.

ניתן ליצור את עקרון השונות בכוח ע"י ***(time quantum) preemption***.

<u>1. תהליכים וסינכרוניזציה:</u>

משאב: כל דבר שעומד לחלוקה או לשיתוף (למשל: יחידת ק/פ, זיכרון, CPU (ה-CPU הוא משאב של זמן), הזיכרון הוא משאב פיזי שעובד בשיטה של מרחב (space). תוכנית ונתונים: משאב לוגי).

תהליכים: (*process, task*)

מע' הפעלה מופעלת מאוסף תהליכים שמתחרים ביניהם על המשאבים. תהליך נשלט ע"י תוכנית מסוימת ודורש זמן (CPU), זיכרון (מרחב) ומשאבים נוספים. *process = <processor, program>*

כשאוסף הפקודות (*program*) מתבצע מקבלים תהליך.

<u>תהליך סדרתי</u> נובע מביצוע תוכנית ועיבוד נתונים ע"י מעבד סדרתי.

מע' ההפעלה היא אוסף תהליכים שחיים באנרכיה. אין תהליך מרכזי. יש תהליכים דומיננטיים, למשל: dispatcher, allocation וכו'.

ניתן לראות כל תהליך כ"מעבדון" קטן. ה-*uniprocessor* הוא סימולציה של *multiprocessor*.

תהליכים יכולים להיות משותפים באותה תוכנית:

- שותפות מלאה: reentrant code – משרתת בבת-אחת כמה תהליכים. לא ניתן לשנות את התהליכים באמצע.
- שותפות לא מלאה: serially reusable code. self modified – התוכנית יכולה לשנות את עצמה, ובסופה התוכנית חוזרת למצבה המקורי.

אפליקציה מבצעת ביצוע סדרתי של הוראות אחת אחרי השניה (למעט הסתעפויות). הסביבה היא תחת בקרה של התוכנית והיא משתנה רק ע"י התוכנית, אלא אם התוצאות דרושות בזמן מסוים. משך הביצוע (*elapsed time*) לא מעניין את התוכנית (הפסקות, חילוף CPU). התוכנית היא יחידה סגורה שרצה ללא תלות והפרעה.

תהליך הוא ביצוע אחד ויחיד של השגרה. כל ביצוע (אפילו של אותו קוד) הוא תהליך אחר לחלוטין. תהליך מתקשר בלי הרף עם תהליכים אחרים. קל מאוד ליצור ולהרוג תהליכים אחרים ויש תלות הדדית. אם תהליך אב יוצר תהליך בן אזי:

1. העדיפות של הבן לא גדולה מזו של האב.
2. אם האב נהרג, גם הבן נהרג.

$OS = \{P_1, \ldots, P_n\} + \{R_1, \ldots, R_m\}$ {אוסף של תוכניות ושל משאבים}

זמן צפייה: לכל תהליך P_i מתקיים:

לכל $t < \tau_{i1}$ ← P_i = not born.

$t = \tau_{i1}$ התהליך נולד ואז: P_i = Run (active) או P_i = ready (waiting).

$t = \tau_{i2} > \tau_{i1}$ התהליך מת: P_i = not born.

תהליך P_i יכול להוליד/לעורר תהליך אחר: $P_i \rightarrow P_j$.

<u>המודל של ברנשטיין</u> חשוב להבנה מהי יכולת הריצה של תהליכים, משמש למחשבים מקבילים לניתוח תלויות בתוכנית ומשמש למהדרים להפוך תוכניות סדרתיות לתוכניות מקביליות.

נגדיר:

קבוצת תהליכים שמהווים חישוב כלשהו: $P = \{P_1, \ldots, P_n\}$.

יחס קדימויות: סדר הביצוע בין תהליכים: $\Leftarrow$

יחס הקדימויות הוא קבוצה על הזוגות הסדורים מתוך המרחב PxP:

$\Rightarrow$ = { (P_i, P_j) | P_i חייב לגמור לפני ש-P_j מתחיל}

אם $(P_i,P_j)\in\Rightarrow$ ו- $P_i \Rightarrow P_j$ אזי P_i _קודם מיידי_ ל-P_j, P_j _עוקב מיידי_ ל-P_i.
אם $P_i \Rightarrow P_j \Rightarrow \ldots \Rightarrow P_k$ אזי $P_i \overset{*}{\Rightarrow} P_k$ – P_i קודם לא מיידי.
2 תהליכים הם ***עצמאיים*** אם אף אחד _לא עוקב_ של השני. **רק תהליכים עצמאיים יכולים להתבצע בו זמנית.**

תהליך מתבצע כשהוא מקבל תאי קלט מהזיכרון M, מחשב משהו וכותב חזרה אינפורמציה לתאי זיכרון.
לכל תהליך P יש domain ו-range כך ש-P קורא מ-D(P) עד הסיום ורושם ב-R(P) – $R(P)\neq\phi$ תהליך חייב להפיק משהו.

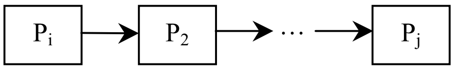
מע' תהליכים נקראת ***דטרמיניסטית*** (***פסקנית***) אם, מבלי להתחשב בסדר הביצוע של P ובחפיפות בביצוע שמוגדרים ע"י יחס הקדימויות, סדר הערכים הנכתבים בכל תאי הזיכרון הוא זהה. המצב הסופי של המע' זהה בכל המצבים.

- ההגדרות לא מאפשרות בדיקה פשוטה האם המע' היא דטר' או לא, כי יש הרבה אפשרויות ביצוע לכל תהליך.

התאבכות הדדית – Mutual Interfering:

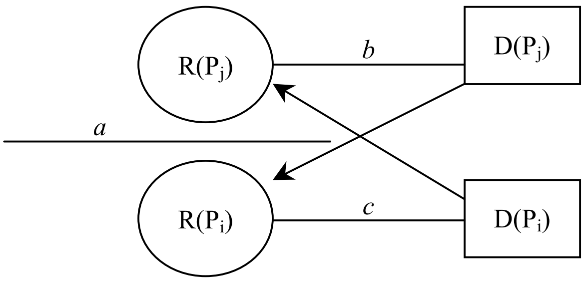
תהליכים שלא מתאבכים הדדית: שני תהליכים P_i, P_j לא מתאבכים הדדית אם הם מקיימים אחד מבין 2 התנאים הבאים:

1. $P_i \overset{*}{\Rightarrow} P_j$ או $P_j \overset{*}{\Rightarrow} P_i$ (אין סיכוי להתאבכות בין הקלט והפלט שלהם).



2. $$\underbrace{[R(P_i)\cap R(P_j)]}_{a} \cup \underbrace{[R(P_i)\cap D(P_j)]}_{b} \cup \underbrace{[D(P_i)\cap R(P_j)]}_{c} = \phi$$

a) - אין חיתוך בין הפלט שלהם, c - אין חפיפה בין הפלט של i לקלט של j, b - אין חפיפה בין הפלט של i לקלט של j.



כלומר, התהליכים הם לחלוטין עוקבים או שאם הם רצים במקביל, שניהם לא כותבים באותה את ל-region של הפלט או שיש חפיפה.

תנאי ברנשטיין:

1. מע' של תהליכים שלא מתאבכים הדדית היא ***פסקנית*** (ולפיכך יודעים שיש יכולת של מקביליות וסדרתיות והתוצאות בשני המקרים תהיינה עקביות).
2. נתונה מע' של תהליכים שבה לכל תהליך P, D(P) ו-R(P) נתונים (יודעים מה מקור הקלט ומה תכולת תאי הפלט), אבל פונקציית המיפוי $f_p: D \rightarrow R$ לא הוכתבה (מיפוי = כיצד לעבור מהקלט לפלט), כלומר: היישום בצורה שאני רוצה.
   אם המע' היא פסקנית **לכל** הפענוחים, אזי התהליכים לא מתאבכים הדדית.

משפט 2 ← אם ניקח קבוצה חלקית של התהליכים הפסקנים, הם יהיו לא מתאבכים הדדית.
המע' יכולה להיות פסקנית לגבי חלק מהתהליכים למרות שחלקם יכולים להתאבך הדדית.

מערכות מקביליות מירביות:

המטרה: למצוא מה הפענוח הטוב ביותר שעדיין יהיה דטר', פסקני ולא מתאבך הדדית.
הגדרה: אם נתונה מע' פסקנית של תהליכים אזי היא קרויה ***מע' מקבילית מירבית*** באם סילוק של זוג כלשהו (P_i, P_j) מיחס הקדימויות גורם לכך ש-P_i, P_j יהיו תהליכים מתאבכים.
כלומר, במע' מקבילית מירבית שכל מקבלים מגבלה על ביצוע חופף ביחס הקדימויות הנוצר הוא באמת הכרחי.
הגדרה: 2 מע' עם אותם תהליכים ויחסי קדימויות שונים נקראות ***אקווילנטיות*** אם באותם תנאים התחלתיים של הזיכרון התהליכים כותבים אותן סדרות של ערכים לתוך תאי הזיכרון ולפיכך מפיקים אותן תוצאות.
הסבר: ניתן להגיע לתוצאות זהות מתהליכים זהים שלא מתבצעים בהכרח באותו סדר קדימויות.

המשפט השלישי של ברנשטיין: בהינתן מע' פסקנית של תהליכים עם יחס קדימויות $\Rightarrow$, נבנה מע' אחרת עם אותם תהליכים ויחסי קדימויות שונים ונקרא לה $\Rightarrow'$. המע' החדשה נבנית בצורה הבאה:

$$\Rightarrow' = \{(P_i,P_j)\in \zeta L(\Rightarrow) \mid ([R(P_i)\cap R(P_j)] \cup [R(P_i)\cap D(P_j)] \cup [D(P_i)\cap R(P_j)] \neq \phi\}$$

מע' זו היא ***מע' מקבילית מירבית היחידה האקוויוולנטית למקורית*** (יכולות להיות מס' מערכות אקוויולנטיות, אך רק מע' מירבית אחת).

Transitive Closure ζL:

(P_i, P_j) הוא חבר ב-$\zeta L(\Rightarrow)$ אם (P_i, P_k) ו-(P_k, P_j) הם איברים ב-$\Leftarrow$.

<u>*האם ניתן להגיד שכל process הוא גם processor?*</u> האם ניתן לראות תהליך כמחשב זעיר?
האם ניתן לראות רכיב תוכנה כרכיבי חומרה?
התשובה היא כן.

עושים סימולציה של מעבד מקבילי על מעבד סדרתי. אפשר גם לראות כל תהליך כאילו היה לו מעבד משלו. הסימולציה אומרת שעושים *context switch* ולכן צריך לשמור *state vector* שבו נתונים חיוניים לתהליך. לאחר מכן, כשמריצים את התהליך מחדש, אפשר להעביר את הנתונים חזרה למקומם, וחוץ מזה שה-*elapsed time* גדל, התהליך לא יודע כלום.

אם תהליכים מתאבכים מתחילים להיכנס ← צריך ליצור ***סינכרוניזציה***.

<u>***אזור קריטי וסינכרוניזציה:***</u>

<u>הבעיה:</u> מס' תהליכים רוצים לגשת לאזור משותף באופן א-סינכרוני ורוצים לשנות את תוכן האזור. יש להבטיח מפאת שינוי סימולטני של 2 תהליכים או יותר.

<u>*דוגמה:*</u>

R1 ← x
R1 ← R1 + 1 ← בצד שני רץ אותו דבר בדיוק.
R1 → x

נניח ש-$x = 4$. אם במיקום החץ יש פסיקה ונכנס תהליך אחר שעושה אותן פעולות, אזי התוכנית תחזיר תשובה שגויה, כלומר: R1 הוא האזור הקריטי.

החלוקה היא ברמה יותר נמוכה של <u>הפקודות</u>. גם פעולה שנראית כלפי חוץ כפעולה אחת, מורכבת מבפנים מיותר פעולות ואם מאפשרים פסיקות באמצע פקודה יכול להיווצר מצב שגוי.
בשביל מה בכלל צריך פסיקה באמצע פקודה? יש פקודות שאורכות הרבה cycles. צריך את הפסיקות למקרי חירום, בעיקר במערכות זמן-אמת. לכן הפקודות אינן אטומיות.

יש איזורים רבים שהם איזורים קריטיים, כלומר:

1. הפקודה חייבת להסתיים כסדרה: MCF – Must Complete Function.
2. הפעולה צריכה להיות מנק' מבט חיצונית אטומית, כלומר: רק אחד יכול להיות בהם. אסור להרשות ליותר מתהליך אחד להיות באזור הקריטי.

לכן:

1. נאסור מקביליות ← אין multi programming.
2. נרשה מקביליות אבל נאסור פסיקות ← יש multi programming.
3. לא נרשה מקביליות רק בכניסה לאזור הקריטי. ברגע הראשון שנסיים את ה-CS נחזיר את המקביליות.

<u>*הנחה:*</u> בקטע הקריטי נתכנת בצורה הטובה ביותר והדחוסה ביותר כך שהפעולות שם תהיינה מהירות.
כלומר: יש מקביליות, קטע קצר של סינכרוניזציה, ושוב מקביליות.
לשיפור: נעשה הגנה נפרדת לכל אזור קריטי.

<u>***5 תנאים לקיום אזור קריטי:***</u>

1. אסור ל-n תהליכים להיות בו-זמנית באזורים הקריטיים שלהם.
2. אסור להניח הנחות כלשהן על מהירויות או מס' מעבדים (*race condition*).
3. אסור לתהליך שנמצא מחוץ ל-CS (בין אם הוא רץ שם או שהוא במצב *ready* או *blocked*) לחסום תהליך אחר מלהיכנס ל-CS.
4. שום תהליך לא ייאלץ לחכות לנצח לרשות כניסה ל-CS שלו.
5. אי-אפשר להצמיד עדיפות לתהליכים וההתייחסות לאזור הקריטי תמיד תהיה א-סינכרונית.

<u>***פתרונות:***</u>

<u>***I. פתרון תוכנה:***</u>

1. הפתרון של פיטרסון – פתרון בתוכנה. בשביל זה נצטרך שהתוכנית תעבוד ללא פסיקות. באופן מעשי, פתרון תוכנה הוא אינו טוב.

פתרון לא נכון:

המשתנה המשותף *turn* מאותחל (ל-0 או ל-1) לפני ביצוע P_i.
הקטע הקריטי של P_i מתבצע אם"ם *turn = i*.

```
Process Pi:
    repeat
        while (turn != i) {/* wait */ };
            CS
        turn := j;
            RS
    forever
```

מתקיימת ***מניעה הדדית***: P_i ממתין אם P_j בקטע הקריטי.
לא מתקיימת ***התקדמות***.
אם תהליך צריך את הקטע הקריטי יותר מאחרים, הוא לא יקבל אותו.
הבעיה: הפעולה מחייבת כניסה סדרתית אחד אחרי השני. אם אחד לא רוצה להיכנס והשני רוצה להיכנס עוד פעם זה יהיה בלתי אפשרי.

2. הפתרון של למפרט - אלגוריתם "קופת-חולים": פתרון ל-*n* תהליכים:

לפני הכניסה לקטע הקריטי, כל P_i מקבל מספר. המחזיק במספר הנמוך ביותר נכנס לקטע הקריטי.
אם P_i ו-P_j מקבלים אותו מספר: אם $i < j$ אזי P_i נכנס קודם, אחרת P_j נכנס קודם לקטע הקריטי.
P_i מאפס את המספר שלו בעת היציאה מהקטע הקריטי.
נסמן:

(a,b) < (c,d) if a < c or if a = c and b < d
$\max(a_0, \ldots, a_k)$ is a number b such that $b >= a_i$ for i = 0, … ,k

```
Shared data:
  choosing: array[0..n-1] of boolean; // initialized to false.
  number: array[0..n-1] of integer; // initialized to 0

Process Pi:
  repeat
        choosing[i] := true;
        number[i] := max(number[0]..number[n-1]) + 1;
        choosing[i] := false;
        for j := 0 to n-1 do {
            while (choosing[j]) {};
            while (number[j] != 0 and (number[j], j) < (number[i], i)) {};
        }
        CS
        number[i] := 0;
        RS
  forever
```

מכריז – choosing = true על רצוני לקבל מס' כדי להיכנס ל-CS.

אח"כ בודקים כל תהליך ואם אף תהליך לא בוחר מספר אזי אני אכנס ל-CS רק אם המספר שלי קטן מהמספר של התהליכים האחרים.

הפתרון הנ"ל עובד אם כל הפונקציה היא ***אטומית***.

היתרון בשיטה זו הוא שהתהליך סוחב איתו את המס' שלו וזה טוב לרשתות תקשורת.

II. פתרון חומרה:

פקודת Test and Set:
פקודת T&S היא פקודה אטומית. בסוף T&S ערך i יהיה תמיד שווה ל-1.
T&S הוא הבסיס לבניית ***סמפורים***. אחרי ה-T&S נבדוק איזה מצב הוחזר: אם הוחזר *true* ← ניתן להיכנס ל-CS, אחרת: נכניס את התהליך לתור, אח"כ נעשה *context switch* וכו'.

```
bool testset(int &i)
{
    if (i == 0) {
        i = 1;
        return true;
    } else return false;
}
```

<u>אלגוריתם שמשתמש ב-*Test and Set* ליצירת ***מניעה הדדית***:</u>
המשתנה המשותף b מאותחל ל-0.
רק התהליך הראשון P_i שקובע את b נכנס ל-CS.

```
Process Pi:
    repeat
        repeat{}
        until testset(b);
          CS
        b := 0;
          RS
    forever
```

<u>בעיה</u>: ***busy-wait***. כאשר הלולאה של ה-*busy-wait* קצרה יותר ממס' הפקודות הדרושות כדי להיכנס לתור במע' ההפעלה, אזי עדיף להשתמש בלולאה.

דיקסטרה:
P ו-V הן פעולות על מס' שלמים לא שליליים שעובדים על סמפורים.

הרעיון: ליצור במחשב פקודה שתבצע את P(S) ו-V(S), שתהיינה פקודות אטומיות.
מגדירים סמפור כללי שערכו שווה 1 וכל מי שקורא ל-P(S) אזי:
1. אם S = 1 הוא יחסיר מ-S אחד והקטע הקריטי יהיה שלו.
2. אחרת, הוא מחכה עד ש-S יהיה גדול מ-0.

```
P(S)    if S <= 0
        then    put process in wait
                /* wait until s > 0 */
        else S = S - 1

V(S)    S = S + 1
```

P(S) גורם או להמשך התהליך או לכניסה למצב ***busy-wait*** שבו:
1. או שיש לולאה תמידית הבודקת האם ניתן להיכנס.
2. או שנכנסים ל"שינה" ותהליך אחר יעיר אותי כשניתן יהיה להיכנס.

אם בתור של הממתינים יש תהליכים שמחכים ו-$S > 0$ אזי פונים למשגר ואומרים לו לקחת תהליך מהתור ולהכניסו ל-CS.

תוצאת V היא פסיקה.

לפני הקטע הקריטי נעשה P(S) ובסיום V(S):

```
global mutex = 1;

Li:      P(mutex)
          CS
         V(mutex)
          RS
         goto Li
```

מניעה הדדית בעזרת סמפור

בעיית היצרן-צרכן:

כשיש יצרן אחד וצרכן אחד:

```
global S = 0
P1:  P(S) /*  הצרכן: אחרי P(S) - מעבד את האובייקט */
P2:  V(S) /*  היצרן: לפני V(S) - יוצר את האובייקט */
```

בדוגמה הנ"ל יוצרים איבר אחד וצורכים איבר אחד.

מכלא מעגלי סופי בגודל k:

אחרי שמכניסים מידע ל-b_{k-1} הבא יהיה b_0.

צריך: סמפור S שיוצר ***מניעה הדדית*** לכניסה למכלא.
סמפור N שיסנכרן בין היצרן לצרכן בנוגע למספר הפריטים הקיימים (עצירת הצרכן כדי שלא ישיג את היצרן) – צריך לדעת כמה אלמנטים מלאים יש.
סמפור E שיסנכרן בין היצרן לצרכן בנוגע למספר המקומות הריקים במכלא – צריך לדעת כמה אלמנטים ריקים יש.

```
Initialization:   S.count := 1; in := 0;
                  N.count := 0; out := 0;
                  E.count := k;

append(v):              Producer:             Consumer:
b[in] := v;             repeat                repeat
in := (in + 1) mod k;       produce v;            wait(N);
                            wait(E);              wait(S);
take():                     wait(S);              w := take();
w := b[out];                append(v);            signal(S);
out := (out + 1) mod k;     signal(S);            signal(E);
return w;                   signal(N);            consume(w);
                        forever               forever
```

פתרון לבעיה:

היצרן בלולאה אינסופית מייצר v. הוא עושה wait(E) ל-E שהוא סמפור מונה. אחרי ה-wait הראשון $E = k - 1$. אם $E = 0$ הוא נתקע.
עכשיו עושים wait(S) כיוון שב-append רק היצרן או הצרכן יהיו בתא. S שומר על האטומיות של append. S הוא סמפור בינארי.

The Dining Philosophers Problem:

5 פילוסופים שרק אוכלים או חושבים. כ"א צריך 2 מקלות בשביל לאכול. יש רק 5 מקלות. בעייה קלאסית של סינכרון שממחישה את הקשיים בהקצאת משאבים בין תהליכים בלי יצירת ***deadlock*** ו-***starvation***.
כל פילוסוף הוא תהליך. לכל מקל יש סמפור.
נוצר ***deadlock*** אם כל פילוסוף מתחיל ע"י הרמת המקל השמאלי.
אי-אפשר למנוע הרעבה מכוונת.

```
stick: array[0..4] of semaphores
Initialization:
        stick[i].count := 1 for i:= 0..4

Process Pi:
        repeat
            think;
            wait(stick[i]);
            wait(stick[i+1 mod 5]);
            eat;
            signal(stick[i+1 mod 5]);
            signal(stick[i]);
        forever
```

פתרון נוסף:
נאפשר רק ל-4 מחמשת הפילוסופים לאכול בו-זמנית.
במקרה הגרוע ביותר: אם 3 מרימים את המקל השמאלי, עדיין ה-4 יוכל לאכול. כשהוא יסיים הוא יניח את המקל והחמישי יוכל לאכול. כיוון ש-4 מתוך ה-5 רשאים להיכנס לקטע הקריטי, אחד תמיד יוכל לאכול. המימוש ע"י סמפור מונה T.

```
stick: array[0..4] of semaphores
Initialization:
        stick[i].count := 1 for i:= 0..4
        T.count := 4

Process Pi:
        repeat
            think;
            wait(T);
            wait(stick[i]);
            wait(stick[i+1 mod 5]);
            eat;
            signal(stick[i+1 mod 5]);
            signal(stick[i]);
            signal(T);
        forever
```

שיטה נוספת: נרחיב את הסמפורים הפשוטים על ידי התניות של and, not, or.
הרחבה מסוג and:
הפסקת שני התהליכים במיקום החצים יוצר *deadlock*.
נשנה את זה ל: P(mutex_x) && P(mutex_y).

$P_R(A)$	$P_R(B)$
P(mutex_x)	P(mutex_y)
→ P(mutex_y)	P(mutex_x) ←

עפ"ר, ההרחבה תהיה:

```
SP(S1, … , Sn)
    if (S1 >= 1 and ,… , and Sn >= 1)
    then for i := 1 to n do
            Si = Si – 1;
         end for
    else
      הצב את התהליך בתור ששייך ל-Si הראשון
        שמקיים Si < 1, והצב את מונה הכתובות
      כך שאם Si משתחרר חוזרים על התהליך.
    end if
```

```
SV(s1, … , sn)
    for i := 1 to n do
        Si = Si + 1;
      ואם Si > 0 אזי כל התהליכים הנמצאים בתור
      Si מועברים למצב ready.
    end for
```

זאת-אומרת, התהליך יתבצע רק כאשר **כל** הסמפורים פנויים (יכול להיווצר מצב שנחכה ל-S_4 שיתפנה. כשהוא יתפנה נצטרך לחכות ל-S_8, אח"כ שוב ל-S_4 וכו').

מותר שתהיה פסיקה, אבל התהליך היחידי שיוכל לחזור ל-CS הוא אני, ולכן מנקודת מבטי, כאילו אין הפסקה.

משתמשים בזה בעיקר לעבודות batch שבהן התוכנית מכריזה מראש את כל המשאבים שהיא צריכה, ועד שאין את כל המשאבים לא נכנסים לביצוע.
הדבר הזה לא יכול להתקיים כשלא יודעים מראש מה צריך.

המודל הזה הוא **בזבזני** – הכל (לחכות המון זמן) או לא כלום.
במצב זה אין deadlock אבל יכולה להיווצר הרעבה.

את בעיית 5 הפילוסופים ניתן לפתור בשיטה זו של סמפור מקבילי:

כל מבנה כזה, מנקודת המבט של התהליך, הוא אטומי (חסום ע"י test and set).

```
global
    chopstick: array[0..4] of Semaphores(1,1,1,1,1)

Pi:
    repeat
        think;
        SP(chopstick[i] && chopstick[i+1 mod 5]);
        eat;
        SV(chopstick[i] && chopstick[i+1 mod 5]);
    forever
```

גישה א': 1. P1 מבקש סמפור.
2. P1 מקבל סיגנל ok.
3. P1 נכנס לסמפור.
בזמן גדול מ-1 P2 ניגש לסמפור ומקבל הודעה שהוא תפוס ← P2 נמצא ב-wait.
גישה א' משתמשת ב***סמפור חיצוני***.
בגישה זו יש אפשרות "לרמות" ולגשת ל-CS ע"י branching בלי לבקש רשות קודם מהסמפור החיצוני.
גישה ב': לפני ה-CS נוסיף חתיכת קוד שיכיל את הסמפור. P1 בזמן 1 הולך למשאב. הוא **לא נכנס למשאב אלא לקוד הבקשה**. אם הסמפור פנוי אזי:
1. הסמפור הופך ללא פנוי.
2. הקוד ממשיך לתוך ה-CS.
3. P1 מקבל status שהוא הצליח בבקשה.
P2 בזמן גדול מ-1 מגיע לקוד הבקשה שרואה שהסמפור תפוס ולכן מחזיר לו wait.
גישה ב' משתמשת ב***סמפור פנימי***.
גישה ב' היא ***monitor***.
בגישה זו יש פחות שגיאות שקורות עקב קפיצה ל-CS בלי קבלת הרשות המפורשת.

רמות של פרימיטיבים:
הפרימיטיב הפרימיטיבי ביותר הוא ***ניטרול כללי***: סגירת כל הפסיקות. רק 2 פסיקות עובדות:
1. SVC/IRQ – פסיקה מתוכנתת.
2. Machine Check – אם משהו מתקלקל.

חסרונות: 1. לא טוב ב-multi programming.
2. לא יעיל.

רמה גבוהה יותר:
1. Test and Set.
2. Compare and Swap.
3. Compare and Swap Double.

ע"י test and set ניתן ליצור רוטינות שאינן חליקות (למשל: oil – or immediate logic, nil – not immediate logic). כעת ניתן לכתוב רוטינות ארוכות – פרימיטיבי פעולה. למשל: lock – כשרוצים לעשות עדכון במסד-נתונים. ה-lock הוא הרבה פקודות ובלתי-חליק.

יש כמה סוגי משאבים:
1. משאבים מתכלים (למשל: נייר מדפסת, קריאת אינפורמציה למכלא ואח"כ שחרור).
2. משאבים שמישים בצורה סדרתית: למשל: כונן סרטים מגנטי, שגרה שמשנה את עצמה ובסוף מסדרת את עצמה.
3. reentrable (שמיש בו-זמנית): למשל: dll.

ENQ/DEQ: בקשת רשות להשתמש במשאבים שהם שמישים בצורה סדרתית.
ניתן להשתמש במשאבים שמישים סדרתיים ב-2 צורות:
1. אקסקלוסיבית (רק אני אשתמש במשאב).
2. shared – אני ואחרים יכולים להתחלק במשאב.

כיצד מגנים על המשאבים? משאב מיוצג ע"י 2 שמות (הסיבה: יעילות). הסכם השמות מכובד ע"י כל המשתמשים.
שיטת ה-bucket sorting: אם הנתונים אותם רוצים למיין מתפלגים ל-k קבוצות הומוגניות, אזי מחלקים אותם ל-k קבוצות וממיינים בתוך הקבוצה.

ENQ – פקודה לבקשת משאב:

$$enq\ qname, rname, \{ \begin{matrix} e \\ s \end{matrix} \}, rname\_length, \{ \begin{matrix} step \\ system \\ systems \end{matrix} \}, ret = \{ \begin{matrix} chng \\ have \\ test \\ use \\ none \end{matrix} \}$$

הדלי הוא qname – "שם המשפחה". גודלו: 8 סימנים (למשל: sys00ios).
rname – "שם הפרטי". גודל משתנה (עד 255 תווים) ולכן יש פרמטר של rname_length.
$\{ \begin{matrix} e \\ s \end{matrix} \}$ - צורת הבקשה: בצורה אקסקלוסיבית או shared.

$\{ \begin{matrix} step \\ system \\ systems \end{matrix} \}$ - הטווח של התהליך. האם השמות qname, rname ידועים רק לתהליך שלי (step); או שהוא ידוע לכל התהליכים במערכת (system); או שהוא ידוע בין כמה מחשבים (systems).

אם לא כותבים ret אזי ביקשתי זכות גישה למשאב כלשהו. אחרת:
שינוי זכויות: change = chang: אפשר שישנו את הדרישה מאקסקלוסיבי ל-shared ולהיפך (השימוש בזה נעשה כשכבר בצענו בקשה (e, s) ואנחנו רוצים לבצע שינוי בלי לשחרר ולבקש שוב).
have = תן את המשאב רק אם לא ביקשתי אותו.
test – לספר מה מצב המשאב: האם ביקשתי או לא ומה מצבו (אקסקלוסיבי או shared).
use – תן לי את המשאב רק אם הוא פנוי (לא נכנס לתור).
none – ברירת המחדל: ללא תנאים.

:DEQ
$$deq\ qname, rname, \left\{\begin{matrix} step \\ system \\ systems \end{matrix}\right\}, ret = \left\{\begin{matrix} have \\ none \end{matrix}\right\}$$
have – מוותר על המשאב רק אם יש לי אותו.

__איך זה מיושם?__
יש מצביע ל-CVT (central vector table). ב-280 + CVT יש עוגן לניהול מע' התורים של בקשות enq/deq.
בפעם הראשונה שיוצאת בקשה למשאב היא בונה 3 control block:
Major QCB – של qname.
Minor QCB – של rname.
QEL – האלמנט שמתאר את הבקשה הספציפית.
כשתהליך נוסף מבקש אותו QCB, אותו qname ו-rname אזי ל-QEL הראשון מצטרף QEL נוסף. אם בשניהם ה-exclusive לא דולק ← שניהם יקבלו גישה במקביל. אם לאחד מהם ה-exclusive דולק אזי השני ימתין אלא אם כן הוא רוצה רק להסתכל ולא לעשות שינויים.
כשעושים DEQ ה-QEL שוויתר על המשאב יוצא מהתור והמע' בודקת את ה-QEL הבא. כשה-QEL האחרון נדרס ← ה-minor QCB נהרס ← ה-major QCB נהרס.
יכול להיווצר מצב שיהיה major QCB שיצביע על הרבה minor QCB.

- לא מספיק יעיל לעשות את הבדיקות בפרק זמן קצר. היום עובדים עם עצים מאוזנים עקב הרבה multi programming.

__מוניטור:__
את כל התוכניות שפונות ל-CS נשים ביחד והמוניטור יקפוץ בין תוכנית לתוכנית (לפי התור). המוניטור מבצע את פעולת הסינכרוניזציה עבור כל התוכניות והוא מונע בכך בצורה טובה התנגשויות.
היתרון הוא ביציבות. המתכנת לא צריך לדאוג לכל הבקשות והפרמטרים, הוא פשוט קורא לפרוצדורה של המוניטור ואז מופעל מה שצריך.
החיסרון: ברוב השפות המוניטור אינו מוגדר היטב ולכן לא משתמשים בו הרבה.

Message Pass: send, receive. גישה טובה למחשב המקומי וברשתות תקשורת בהן קשה לקחת סמפור משותף: receive(source, msg),   send(dest, msg).
המסר משמש ל: 1. סינכרוניזציה.   2. הודעה.
ה-receive יקבל מידע וישים אותו בתוך ה-msg. עפ"ר, ה-send אינו blocked. ה-receive הוא במצב blocked – עד שאני לא מקבל את המסר המתאים אני ממתין. אחרי ה-receive יש send:
1. אפשרות לעצור send (שנכנס למשל ללולאה אינסופית).
2. אפשרות ל-ack.
3. בקשה לשליחת ההודעה מחדש.
ז"א, כאשר רוצים לבצע פעולה על CS ויש חשש שאסור לבצע זאת מכיוון שזה משאב משותף, אזי מוציאים עליו פקודת receive וממתינים עד לקבלת send.
יש שיטה שאומרת שאת ה-send שולחים לקופסה שמכילה port או mailboxes. ב-port כל המספרים עוברים לערימה כוללת. בתחילת כל מסר יש header שאומר לאיזה port להיכנס. ב-mailbox בונים תיבות ומכניסים את המסר לתיבות, והשולף לוקח את המסר מהתיבה שלו.

__:Locks__
פונים למשפחות שונות של סינכרוניזציה.
יש היררכיה בפקודות, לכן מחלקים את הפעולות לגלובליות ולמקומיות.
כשנכנסים למשאב גלובלי ולא מקבלים אותו אזי נכנסים למצב של busy-wait.
כשנכנסים למשאב מקומי ולא מקבלים אותו אזי נכנסים למצב של suspend ← המשאב נכנס ל-blocked queue, כלומר הוא מוותר על הריצה שלו והפיקוד עובר ל-*dispatcher* שיחליט מי ירוץ.

נוצרו 2 פעולות אקוויוולנטיות ל-P ול-V של setlock שהיא בקשה ממע' ההפעלה לתת lock בהתאם למפורט.
$$setlock\ \ obtain, type = \quad , mode = \left\{\begin{matrix} cond \\ uncond \\ uncond\ \ disabled \end{matrix}\right\} \qquad [equivalents\ to\ P]$$
$$setlock\ \ release, type = \quad , mode = \left\{\begin{matrix} cond \\ uncond \\ uncond\ \ disabled \end{matrix}\right\} \qquad [equivalents\ to\ V]$$

type – סוג ה-lock.
cond – תן לי אותו רק אם הוא פנוי (כלומר: אם הוא לא פנוי אני מוותר ולא נכנס לתור).
uncond – תן לי אותו אם הוא פנוי, אחרת הכנס אותי לתור.
uncond disabled – כמו uncond עם הוספת תנאי: כשאני מקבל אותו תעשה disable לכל הפסיקות.

<u>סוגי ה-type</u>: קבעו מע' היררכית בצורה הבאה:

		#lock	
1	dispatcher	1	
2	asm (address space manager)	4 + #process	איפה שבונים את הטבלאות להתחיל בתהליך.
3	salloc	1	storage allocation – לקבל את הזיכרון
4	ioscate	1 per CPU	לכל קובץ שמור באיזה דיסק ניתן למצוא אותו
5	iosucb	1 per UCB	unit control block
6	ioslch	1 per LCH	logical channel
7	srm	1	מוניטור הביצועים
8	local	1 per as	

עכשיו קבעו חוקי היררכיה: הכיוון הוא תמיד כלפי החץ – הבקשה היא תמיד לשלב יותר גבוה בהיררכיה.
אם יש lock שיש מופעים רבים שלו מותר להחזיק רק באחד.
את הפקודה setlock הפכו למעגל אלקטרוני.

<u>משאבים:</u>
<u>בעלי שימוש חוזר</u>: משאב שבעת שגומרים לעבוד בו הוא משחזר עצמו. יש מס' סופי שלהם.
<u>consumable</u>: אין מס' קבוע של יחידות. אין עניין להשתמש במכלא שמישהו אחר לקח.
<u>reentrant</u>: קוד שלא מלכלך את עצמו. אם יש צורך באזורי ביניים, הוא לוקח איזורי עבודה אחרים.

<u>מועד הקישור – binding time:</u>
מועד הקישור זה הרגע בו נקבע סופית האירוע – פעולת המיפוי.
מועד הקישור הכי מוקדם הוא בעת ה*<u>קומפליציה</u>* (סטטי).
יש מועד קישור בעת הריצה של כניסת השיגרה – malloc at entry (דינמי).
מועד הקישור הכי מאוחר הוא בעת הריצה, אם צריך – malloc when needed. גמיש מאוד אבל יקר מאוד.

<u>יתרון קישור סטטי:</u> מבחינת טעינה וטיפול הוא הפשוט ביותר. יש בזבוז זיכרון, אולם אין בזבוז CPU בעת הריצה ולכן הוא המהיר ביותר.
<u>חסרון קישור סטטי:</u> אינו גמיש: כדי לשנות אותו חייבים לחזור ל-source, לשנות אותו ולעשות קומפילציה מחדש.

<u>Deadlock (קיפאון):</u> נובע בעיקר מכך שעושים את ה-binding time מאוחר.
עובדים בצורה אנרכיסטית. אין control מרכזי.
<u>יתרונות</u>: 1. אין צוואר בקבוק.
2. כל זמן שאין התנגשויות יש אוטונומיה מוחלטת לכל תהליך.
<u>חסרון</u>: כשרוצים משאבים סדרתיים יכול להיות מצב של התנגשות שאינה פתירה.
אם זה זיכרון או CPU ← קל לפתרון. אם db או שאילתות תלויות ← נורא קשה לפתור.

<u>עד כמה זה מפריע?</u>
טריקים לפתרון:
מע' שבה יש מס' סרטים מגנטיים (serially reusable). לדוגמה: יש 4 סרטים ואני מבקש את יחידה מס' 2. אם מישהו אחר גם מבקש את יחידה 2 אזי יש קונפליקט. במקום לבקש יחידה אחת נבקש את כל היחידות בעלות אותו ***אופי*** ואז למע' ההפעלה יש יותר גמישות בבחירה.
<u>בעיה</u>: TAPE = {T_1, T_2, T_3, T_4}, DISK = {D_1, D_2, D_3}. אני רוצה לעשות מיון. דיסק הוא יחידה שיכולה להיות גם יחידה סדרתית. למה שלא נגדיר קבוצה: *serial* = {T_1-T_4, D_1-D_3} ואם צריך למיין וצריך 4 יחידות נבקש 4 יחידות מקבוצת ה-*serial*.
אם ההקצאה נעשית בצורה לא נכונה לפעמים לא ניתן לקבל יחידות.
- אם יש חפיפה (אפילו חלקית) בין הקבוצות הסיבוכיות עשויה להיות גדולה.

1. האם ניתן למנוע מראש deadlock?
   א. האם יש שיטה יעילה של חלוקת משאבים שתמנע בכלל אפשרות ל-deadlock?
   ב. האם יש שיטה להקצאת משאבים מבוקרת כך שבכל רגע נתון אם אנחנו מתקרבים ל-deadlock נוכל לעצור ולהציל את המצב?
2. מה הדרך להשתחרר מ-deadlock ללא עלויות גבוהות מדי?

Swapping: יש חסרון דרמטי בזיכרון. ניקח את ה-job כמו שהוא ונזרוק אותו לדיסק ← פתרון למנוע חנק בזיכרון.
לא צריך את זה בזיכרון וירטואלי כי שם יש תמיד דפים. כן צריך את זה למקרה שטעיתי ונוצר מצב שכל תוכנית מגיעה למצב של thrashing ואני רוצה להקל את היחס בין מס' הדפים הפיזיים למס' הדפים הלוגיים ← נזרוק job שלם לזיכרון.

נניח שיש 2 תהליכים:

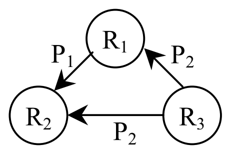
```
A          B
if         if
send       recv
recv       send
```

התוכנית נכתבה נכון אבל בגלל איזשהו תנאי קפצתי לנקודה לא נכונה בתוכנית. כלומר, deadlock לא נגרם רק בגלל הקצאה אלא גם עקב טעויות בתוכנית. פתרון: time out.

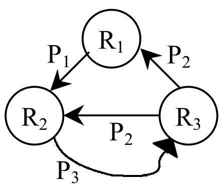
<u>**Deadlock קורה אם מתקיימים 4 תנאים:**</u>
1. ***mutual exclusion*** - התהליכים דורשים שליטה בלעדית על המשאבים שהם ביקשו.
2. ***wait condition*** – התהליכים המשיכו להחזיק במשאבים שהם קיבלו בעת שהם המתינו למשאבים נוספים.
3. ***no preemption*** – לא ניתן לנתק בכוח משאבים מהתהליכים שמחזיקים בהם, וזאת עד שהתהליכים מסיימים את פעולתם.
4. ***circular wait*** – קיימת שרשרת מעגלית של תהליכים כך שכל תהליך מחזיק במשאב אחד או יותר המבוקשים ע"י התהליך הבא בשרשרת.

<u>***אפשרות נוספת להציג deadlock היא באמצעות גרפים:***</u>
$\{P_1, \ldots, P_n\}$ – קב' תהליכים, $\{R_1, \ldots, R_m\}$ – קב' משאבים נפרדים.
נגדיר גרף מכוון:
צמתים = משאבים.
קשתות = אם בפרק הזמן שמתייחס לגרף תהליך P_i מחזיק במשאב R_j בעת שהוא מבקש את R_k, אז הגרף יכלול קשת מ-R_j ל-R_k (הקשת = הבקשה).

<u>דוגמה:</u>



P_2 מחזיק את R_3 ומבקש את R_1 ואת R_2.
P1 מחזיק את R_1 ומבקש את R_2.
אין deadlock – תנאי 4 אינו מתקיים.
אם ניתן קודם את R_2 ל-P_2 אזי יהיה deadlock.



יש deadlock.

לפני כל בקשה נבנה גרף ונחליט אם לאשר או לא. מעגל של loop סגור הוא תנאי הכרחי ומספיק כדי שיהיה deadlock במעגל (בתנאי ש-3 התנאים האחרים מתקיימים גם).

<u>**בעיות:**</u>
1. <u>זמן</u> לבניית הגרף.
2. אם יש יותר ממשאב אחד מכל סוג אזי ציור הגרף הוא תנאי הכרחי ולא מספיק.

<u>***מניעת Deadlock:***</u>
1. ***השיטה האבסולוטית:*** המשתמש יגדיר את כל המשאבים שהוא צריך בעת הצגת התוכנית למע' וה-dispatcher לא ישגר שום job עד שהמשאבים יתפנו.

<u>*חסרונות:*</u> 1. בזבוז: צריך לחכות עד שהכל יהיה זמין ואין צורך בכל המשאבים ביחד. אין מצב של שחרור מוקדם של משאבים.
2. אי-ידיעה מראש של המשאבים.

<u>*טריקים:*</u> נכריז מראש מה אני רוצה. השאלה: מתי אני משחרר את המשאבים. ניתן לשחרר את המשאבים ברגע שעשיתי את השימוש האחרון שלו.
בהקצאה האבסולוטית זמן הקישור מוקדם מאוד וזול מבחינת הטיפול, אבל לא גמיש.

2. ***השיטה היחסית:*** מודיעים כשרוצים. כאן ניתן להיכנס ל-deadlock.

טיפול ב-Deadlock:

1. ***מניעה – prevention:*** יצירת מע' שמונעת כניסה למצב deadlock.

כדי למנוע deadlock צריך לשבור את אחד מ-4 התנאים ל-deadlock.

1. mutual exclusion: לא ניתן לשבור תנאי זה, אך אפשר אולי להקטין אותו, למשל ע"כ שלא נבקש סמפור לאורך כל חיי ה-job או שנבקש סמפור על חלקי דיסק ולא על כל הדיסק וכיו"ב. ז"א, נצמצם את אזור הקטע הקריטי מתוך תקווה שיהיו לנו פחות התנגשויות.
2. wait condition: אם אני רוצה משאב נוסף ואני מחזיק כבר במשאבים אני חייב לשחרר את כל המשאבים שיש לי ולבקש מחדש.
3. no preemption: משאירים את זה בידי מע' ההפעלה. לרוב לא ניתן למנוע את זה.
4. circular wait: ניתן לכל סוג משאבים מס' סידורי: המקום הראשון – הדיסקים המהירים וכן הלאה. כל job שיש לו משאבים מסוג R_i יכול לקבל רק את אלו שהם R_{i+n} (ניתן רק להתקדם ברשימה: הכיוון – מהטובים יותר לטובים פחות). אם רוצים את R_{i-n} חייבים לשחרר את R_i ולבקש מהתחלה ← לא יכול להיווצר מעגל. לא אידיאלי, כי אם לקחתי בסדר לא נכון אזי צריך לוותר על הכל.

2. ***detection and recovery:***

- נכנסנו ל-deadlock, איך יודעים שאנחנו ב-deadlock?
- אם הגענו למסקנה שאנחנו ב-deadlock איך נמצא מי אשם בזה שאם נהרוג אותו ייפתר ה-deadlock?
- אם אני מצליח להרוג, האם ה-job מרשה לי לחזור ל-checkpoint האחרון שלו?

הבעיה בגרף עבור deadlock: פשטני מדי. רק עבור מצב בו לכל משאב יש מופע אחד.

שיטה שונה:

האלגוריתם של קאופמן:

יהיו $r_1, \ldots, r_s$ משאבים. לכל משאב יש $w_1, .., w_s$ כמויות (מס' היחידות שיש מכל משאב). בזמן מסוים t מתבוננים במע' ומציינים במטריצה P_{ij} את מס' המשאבים מסוג j השייכים לתהליך T_i, כלומר מס' המשאבים שחולקו לו כבר. במטריצה Q_{ij} מציינים את מס' המשאבים מסוג j המבוקשים ע"י T_i.

P – מטריצת ההקצאה. Q – מטריצת הבקשה.

השורה P_i זה וקטור כל המשאבים שברגע הנתון t שייכים לתהליך T_i.

השורה Q_i זה וקטור כל המשאבים שברגע הנתון t תהליך T_i מבקש בנוסף.

$v_1, \ldots, v_s$ – מס' המשאבים הפנויים ברגע t.

תמיד מתקיים $v_j \le w_j$,
$$v_j = w_j - \sum_{i=1}^{n} P_{ij}$$

$\underline{0}$ מסמן וקטור שורה שכל אחד מאיברים שווה ל-0. $\forall i \;\; x_i \le y_i \Leftrightarrow \underline{x} \le \underline{y}$

מטרת האלג': לסדר את כל התהליכים שעדיין צריכים להשלים את העבודה שלהם ולראות מה קורה איתם: האם ניתן בביטחון לתת למשאב ה-i שמחזיק כבר במשאבים את הזמן שלו ולהיות בטוח שהוא לא ייתקע ב-deadlock ולכן להיות בטוח שהוא יגמור ויחזיר את המשאבים ל-pool הכללי?

<u>האלגוריתם:</u>

1. התחל ב-U = v(t) – מציב בוקטור U את וקטור המשאבים הפנויים ברגע t.
נסמן את כל השורות שלגביהן קיים $P_i(t) = \underline{0}$ (מניחים כי כל השורות הן בלתי-מסומנות עם תחילת הבדיקה).
2. חפש שורה לא מסומנת (שורה שכבר מחזיקה משאבים), נניח שהיא i, כך ש- $Q_i(t) \le U$ [כלומר, לא מבקשת יותר משאבים ממה שנמצא כרגע].
אם מצאנו שורה כזו, הולכים לצעד 3.
אחרת, גומרים את הסריקה (צעד 4).
3. הצב: $U = U + P_i(t)$ (כי תהליך זה יסיים ולכן ישחרר את משאביו).
סמן את השורה ה-i.
חזור לצעד 2.
- בעצם אישרתי את הבקשה ואני מעדכן את וקטור המלאי.
4. קיים deadlock אם"ם יש שורות לא מסומנות בעת הסיום וקבוצת השורות האלו מעורבות ב-deadlock.

כלומר, האלג' פותר לנו 2 בעיות: האם יש deadlock ומי אחראי ל-deadlock. הוא לא פותר לנו את הבעיה כיצד להשתחרר מה-deadlock.

האלג' פרופורציוני למס' התהליכים.

גיליתי שנוצר deadlock איך פותרים את זה?

1. נהרוג את התהליכים לפי סדר (מהראשון או מהאחרון).
2. נעשה על יבש תרגיל של הריגה. בכל "הריגה" נוסיף את המשאבים שהתפנו ל-U. אם הוא פותר את האחרים ← נהרוג אותו וסיימנו. אחרת, ננסה להרוג מישהו אחר. בשביל זה צריך לדעת את אופי התהליך:

א. לא הורגים בפראות אחד אחרי השני.
ב. רצים על האלג' ויוצרים קבוצות מינימליות שאם יהרגו אותם ה-deadlock ישתחרר ואח"כ בודקים את הקבוצות מבחינת כמה נזק יגרם. צריך לקבוע מחיר עבור הריגה של job_i.

3. ***התחמקות*** – ***avoidance:*** מסתמכים על ידע של אירועים בעתיד ולאור האירועים העתידיים אני אגביל את גמישות ההקצאות השונות.

נגדיר שכל תהליך מורכב מתהליכי משנה. ההנחה: בעת תת-תהליך השימוש במשאבים נשאר קבוע. בתהליך עצמו יש נקודות של הקצאה ושחרור משאבים. אנו רוצים לדעת האם ניתן לתת משאבים לתת-תהליכים ← גרעיניות טובה יותר ← יעיל יותר.
מגדירים ***צעדים בטוחים (safe steps) S(t):***
ברגע t, בהנחה שאנו נמצאים כרגע במצב בטוח, נתייעץ במטריצות P ו-Q ובאלגוריתם וננסה למצוא סדרה valid כך שכל המע' תסיים את העיבוד.
ברגע ההתחלה אנו ב-safe state. אם יש job שהדרישות שלו קטנות או שוות ל-w אזי ניתן להריץ אותו ← הוא הצעד הראשון. עכשיו נקפוץ ממצב בטוח אחד למצב בטוח אחר. במקרה הגרוע ביותר כשאנחנו פוחדים להתקדם – נתקדם באופן סדרתי, ולכן יש לפחות סדרה טריוויאלית אחת של צעדים בטוחים.
כלומר, ניתן למנוע deadlock אם בכל רגע של מעבר של safe state אני עובר ל-safe state אחר.

<u>מודל נוסף:</u> דרישות המשאבים לא ידועה מראש ולכן לא ניתן לעבוד בתת-תהליכים. אני יודע את המס' המירבי של המשאבים שהתהליך ידרוש (***high watermark***). ***המצב הבטוח:*** סדרה שבה יש תהליכים שהתחילו לעבוד אבל לא נגמרו, כך שכשהם יגמרו הם יוכלו לספק ל-job הבא את הדרישה המקסימלית שלו.

<u>הקלה על המצב:</u> מנסים לקחת משאבים שהם serially reusable ולהפוך אותם למשאבים שהם reentrant (למשל: spooling).
<u>חסרון בשיטה זו:</u> בזבוז CPU, בזבוז פעילות על ה-channel, הזיכרון עולה ביוקר.

<u>Starvation:</u> מצב שבו כל הזמן קיים תהליך אחד או יותר שחסום ומחכה למשאבים שישנם, אך הוא לא מקבל אותם.
<u>פתרון:</u> ***Aging***.

<u>ביזור:</u> 4 מוטיבציות לבניית מע' מבוזרת:
1. ***שיתוף משאבים:*** תהליך במחשב A רוצה לעשות שימוש במשאבים שבמחשב B (משאב = קובץ, חומרה, מסדר נתונים, מדפסת וכו').
2. ***האצת חישובים:*** חילוק חישוב המשימה לתת-משימות. השאלה היא מהי תת-משימה (job שלם? תהליך? חלק מתהליך?), מהי יחידת העבודה הקטנה ביותר שכדאי לעבוד איתה (גרעיניות)?
3. ***אמינות:*** אם יש מתקנים אוטונומיים, נפילת אחד מהם תאפשר לבצע את העבודות באתרים אחרים.
4. ***תקשורת:*** החלפת מידע בין משתמשים שונים (דוא"ל).

<u>איך מחברים את המחשבים?</u> יש 3 פרמטרים:
1. ***מחיר:*** מה המחיר הבסיסי של הקישור?
2. ***מהירות הקישור:*** כמה זמן לוקח לשלוח הודעה?
3. ***אמינות:*** אם אחד מהמחשבים או הקשר נופל, האם השאר יכולים להמשיך לתפקד כרשת.

1. ***חיבור מלא:***
המחיר גבוה מאוד. יש $(n^2 – 1)/2$ חיבורים.
מהירות השיגור גבוהה כיוון שאין תחנות ביניים בין מחשב למחשב.
האמינות מירבית. הרבה מאוד צמתים/קשתות צריכים ליפול כדי להפריד את הרשת.

2. ***חיבור חלקי:***
מחיר נמוך יותר.
תקשורת איטית.
אמינות נמוכה מאוד. לכן דואגים שכל צומת יהיה מחובר ל-**2** צמתים (האינטרנט מחובר בצורה דומה).

3. ***מבנה הירכי:*** קיים בחברות גדולות עם מחשב מרכזי. המחשבים בכל רמה דועכים (A החזק ביותר, B ו-E חלשים יותר).
מחיר נמוך.
אמינות נמוכה. ניתן לבנות חיבור נוסף כדי ליצור ביטחון.
רצוי שהקו בין המחשב המהיר למחשבים האיזוריים יהיה מהיר. מהמחשבים האיזוריים למקומיים אפשר קו איטי יותר.

4. ***שיטת הכוכב:***
מחיר לינארי – זול.
אם C לא צוואר בקבוק ← מכל צומת לצומת אחר יש מקסימום 2 קפיצות ← מהיר יחסית.
אמינות גבוהה. המגרעת היחידה היא אם C נופל. פתרון: שמים 2 מחשבים C.

5. ***טבעת:*** כל צומת מחובר ל-2 שכניו ← צריך 2 ניתוקים כדי לבודד את הרשת ל-2 חלקים.
עלות: לינארית ב-n.
אמינות: די גבוהה.
תקשורת: איטית. תלוי אם הרשת דו-כיוונית או חד-כיוונית. בדו-כיוונית ניתן להגיע עד n/2 שידורים כדי להגיע לצומת ← לכן עושים 2 טבעות: כל אחת רצה בכיוון נגדי.

6. ***Multi Access Bus:*** טבעת שהצמתים נתלים עליו. הקשר לא עובר דרך המחשב. מס' הקפיצות טוב יותר מאשר בטבעת.
עלות: גבוהה יותר מטבעת.
ברשת זו קל מאוד להוסיף ולהוריד ירידות לעומת בטבעת.

היום העיבוד של המחשב נופל מזה של הרשת.

<u>צורת ניתוב השיחות:</u>
אם בין A ל-B יש רק מסלול אחד – עוברים עליו.
אם יש כמה מסלולים יש 3 שיטות:
1. הולכים על נתיב אחד ***קבוע***: אם הוא נופל נעבור לנתיב אחר (אין חיפוש של המסלול הטוב ביותר).
2. ***מסלול וירטואלי:*** כל פעם נבחר מסלול אחר. אחרי שקבעתי אותו אזי לכל אורך ה-session אני עליו. עפ"ר, בחירת המסלול היא לפי המסלול הפחות עמוס.
3. ***ניתוב דינמי:*** בוחרים כל פעם את המסלול הטוב ביותר בין המסלולים. כל packet יכולה לרוץ על מסלול אחר.
<u>חסרון</u>: ריצה על מסלולים שונים ← ה-packets יכולים להגיע בסדר שונה ← צריך packet switching וכו' ← לא טוב לאפליקציות קול.

<u>3 אסטרטגיות לחיבור:</u>
1. ***מיתוג מעגלים – circuit switching:*** יוצרים קשר פיזי בין 2 תהליכים לכל זמן התקשורת. שום תהליך אחר לא יכול לגשת לקשר הזה.
2. ***מיתוג הודעות – message switching:*** נוצר קשר זמני בין התהליכים. הקשרים הפיזיים מוקצים דינמית לפי הצורך. אם יש ניתוב דימני ← מיתוג ההודעות יהיה טוב.
3. ***מיתוג חבילות – packet switching:*** דומה למיתוג הודעות. מיתוג חבילות הוא בגודל משתנה, בעוד מיתוג הודעות הוא בד"כ בגודל קבוע.
4. ***מיתוג תאים – cell switching:*** ב-ATM: תמיד 53 בתים: 48 של data ו-5 של control.

אם עובדים עם bus ← משאב משותף ← קטע קריטי ← 1. בלי סמפור. 2. עם סמפור.

<u>collision detection:</u>
1. ***בלי סמפור: (Ethernet)***
בודקים אם אין כרגע ברשת דו-שיח. אם רץ מסר אני לא נכנס. אם ה-bus נקי אני נכנס. אם באותה שנייה בדיוק נכנס אותו מסר ← יש התנגשות ← ***recovery:*** שנינו נסוגים, כל אחד מגריל מס' רנדומלי ואחרי מס' זה הוא נכנס.
<u>חסרון</u>: יכול להיווצר מצב של הרעבה. 2 תהליכים נכנסים יחד, ולכן הם נסוגים ומגרילים מספר. ברגע שהמספר שלהם מגיע – רץ תהליך אחר.

2. ***עם סמפור:***
token passing: יש הודעה מיוחדת, אסימון, שכל הזמן רצה ברשת. אם צומת רוצה לשגר הוא חייב לחכות שהאסימון יגיע ואז הוא לוקח את האסימון. בסיום השידור הוא מחזיר את האסימון. מי שאין לו אסימון לא יכול לשדר.
מה אם האסימון הולך לאיבוד? מחכים זמן קבוע ואז מגלים שאין אסימון ומישהו מוסיף אסימון חדש. אם שניים הוסיפו אסימון יחד – יהיה אחד שיסלק את הנוסף.
<u>הרחבה:</u> ***<u>message slots</u>:*** במקום אסימון שרץ יש הרבה חריצים. שמים את ההודעות בחריץ פנוי. כשהחריץ מופיע מול היעד מוציאים את ההודעה. ההודעות הן בגודל קבוע.

בטיחות: ב-Ethernet כל הזמן מאזינים, אין פרטיות והכל גלוי. כדי שיהיה סודי צריך להצפין ← עלות.

סוגי מערכות:

1. LAN. 2. WAN.

הסיבה ל-LAN ול- WAN היא כלכלית.

עפ"ר ה-WAN הוא חובק עולם. הדרישות שונות, למרות שלאחרונה זה מתחיל להיטשטש (למשל: ATM הוא גם LAN וגם WAN).

MAN הוא באמצע.

ARPANET: הרשת הראשונה.

<u>קבצים:</u> 1. ***שקיפות:*** האם המשתמש פונה לכל הקבצים במע', בין אם הם מקומיים ובין אם הם במחשבים אחרים, באותה צורה ולא חשוב היכן הם?

2. ***מיקום:*** איפה שוכנים הקבצים עצמם?

<u>I. פרוטוקול העברה של ARPANET:</u>

FTP: לכל מתקן יש קבצים מקומיים. כדי להוריד קובץ אני חייב לדעת בדיוק איפה הקובץ ובאיזה מחשב הוא יושב. חייבים להעתיק את הקובץ אלי כדי לעבד אותו. אין שקיפות ואין שיתוף.

איך מוצאים קבצים? Archie: שרת שמקטלג איפה נמצאים הקבצים.

<u>II. גישה מרכזית:</u>

לכל מתקן יש מע' קבצים שלו וניתן לגשת אליהם רק ממע' המחשבים שם. אבל, כל הקבצים נמצאים ב-file server. יש שקיפות: אם הקובץ אצלי ← אני לוקח אותו, אם לא אזי תישלח בקשה בתקשורת ואני לא אדע איפה זה יושב.

<u>בעיה:</u> יש תלות במרכז ← ביצועים, תחזוקה, אמינות.

<u>III. גישה מבוזרת:</u>

אין file server אבל אני יודע על הקבצים שבמחשבים אחרים. איך?

1. <u>מוותרים על השקיפות</u>: מכניסים את שם המחשב והנתיב (telnet).
2. <u>שומרים על השקיפות</u>: אין מחשב מרכזי אבל מחזיקים ***קטלוג*** שהוא mirror של כל הקבצים של המחשבים.

שיטת NFS:

ב-NFS כל read מביא בלוק מהדיסק המרוחק אלי. יחידת המעבר על גבי הרשת היא בלוקים.

ב-AFS יחידת המעבר היא כל הקובץ (FTP דומה ל-AFS).

<u>ההנחות שעמדו מאחורי פיתוח AFS:</u>

1. קווי התקשורת משתכללים ולכן זה לא נורא להעביר קובץ שלם.
2. אמינות: לאחר שהעברנו את הקובץ והוא התיישב לוקלית אין יותר תעבורה ברשת ולכן אם הרשת נופלת אין לזה השפעה.
3. הקובץ יושב לוקלית במחשב שלי ולכן עיבוד של כל רשומה מהיר יותר.

<u>איך אני משתמש במשאבים משותפים?</u>

1. ***נדידת נתונים:*** מחשב A רוצה לפנות לנתון במחשב B: התוכנית הפונה נשארת ב-A אבל המידע זורם ממחשב B ל-A והעיבוד נעשה ב-A.

<u>חסרונות:</u>

א. עבור קובץ גדול האינפורמציה זורמת המון זמן.

ב. כאשר מעדכנים קובץ לא ברור שמותר להחזיר אותו כי אולי הקובץ היה read only.

ג. עבור כל שינוי קטן שרוצים לבצע יש צורך לקחת ולהחזיר קובץ שלם (אם היחידה המינימלית היא file).

2. ***נדידת החישוב:***

I. נשלח את התוכנית מ-A ל-B בהנחה שהמחשבים הומוגניים (כלומר: שהמחשב ה-2 יודע לבצע את התוכנית, אותה פלטפורמה). נעשה בעיקר למטרות ***load balancing***.

<u>דוגמה:</u> קומפילציה של קובץ ++C בו יש 10,000 שגרות. אני ברשת עם 100 מחשבים. למה שלא ניקח בלוקים של 100 רוטוניות, אבצע אותם במחשב אחר ובסוף נאחד את הכל למחשב שלי.

II. המחשב האחר הוא הטרוגני – ***RPC – Remote Procedure Call***: התוכנית מקומפלת גם על מחשב A וגם על B. מחשב A שולח ל-B בקשה לעורר את התוכנית הרדומה ב-B.

3. ***נדידת עבודות:*** שולחים גם את העבודה וגם את הנתונים למחשב אחר.

<u>מטרה:</u> האצת חישובים במקביל.

החומרה במחשב השני מתאימה יותר.

התוכנה נמצאת רק במחשב השני.

ניתן ליישם עם שקיפות (DCE) או בלי שקיפות.

אין זיכרון/שעון משותף. איך מחליטים איזה מאורע יהיה קודם?

ב-DCE יש שעונים מיוחדים שהם מסנכרני זמן. הם כל הזמן מחליפים מסרים עם מחשבים אחרים ובודקים מה הסטייה של השעון שלהם ומעדכנים:

1. המסנכרנים קשורים כל הזמן לשעון אטומי.
2. קונים שעון שמאזין לתדר רדיו ולפי זה עושים סנכרון.

מנסים לפי הזמן לסנכרן מאורעות ע"י ***סמפור*** ששולח מסרים: אני מודיע לאחרים שאני רוצה להיכנס לקטע הקריטי: אם כולם מאשרים אז בסדר, אחד אומר לא – אז אני לא נכנס. מי ייכנס?

א. לפי השעון. כשיוצאים מודיעים לכולם שהקטע הקריטי פנוי.
ב. לפי אסימון שעובר מאחד לאחד.

חסינות: רוצים להגן על מע' התקשורת מכישלונות. המחשבים מודיעים בהפרשים קבועים שהם חיים. אם מחשב לא מודיע ← קרה לו משהו או שהמסרים הלכו לאיבוד. בכל מקרה צריך לעשות recovery ע"י פנייה אליו: אם מחשב A גילה שמחשב B נכשל, חובה על A לבשר לשאר המחשבים ש-B נכשל (יש פה הרבה overhead: הודעות "חי" וכו'). מחשב שמתאושש חייב להודיע לכולם שהוא זמין.

:DCE – Distributed Computing Environment

מע' גדולה שעושה יישום של הדברים הנ"ל. המע' נבנתה כחלק מ-OSF (Open System Foundation).

DCE מבוססת על: 1. שרת לקוח.
2. RFC
3. data sharing.

ניתן להשתמש ב-WAN או ב-LAN.

DCE מוסר מס' כלים:

1. כלי לפיתוח יישומים מבוזרים.
2. מכלול שירותים שלם: הדרישות לניהול המע' וכו'.
3. עובד ע"י RFC כדי שיוכל לעבוד עם מע' הטרוגניות.
4. תומך בשיתוף נתונים חזק (מילונים משותפים, שירותי קבצים משותפים וכו').

ה-DCE יכול להתקשר למחשבים אחרים שלא שייכים ל-DCE.

__שרת:__ עומד כל הזמן ומאזין אם יש request. שרת יכול להשתמש בשירותים של שרת אחר.

:Daemon תוכנית שמתחילה וכל הזמן מאזינה לבקשות.