Chapter 1:

An **OS**, as opposed to Microsoft belief, is **a** program that acts as an intermediary between a user of a computer and the computer hardware. OS is the first thing that runs on the computer. There is no direct approach to hardware. Every job that requires hardware is going through the OS via system calls.

OS Goals:
- Execute user programs and make solving user problems easier.
- Make the computer system convenient to use.
- Use the computer hardware in an efficient manner.

Kernel: The one program ready to run at all times.

Spooling:

Overlap I/O of one job with computation of another job. While executing one job the OS:
- Reads next job from slow input device into a storage area on the disk (job queue).
- Outputs data of pervious job from disk to slow output device (i.e: printer).

Job pool - data structure that allows the OS to select which job to run next in order to increase CPU utilization.

סוגי מערכות הפעלה:

Time-Sharing Systems - Interactive Computing:

The CPU is multiplexed among several jobs that are kept in memory and on disk (the CPU is allocated to a job only if the job is in memory).

When there is no space in the memory - "Virtual Memory" - a job is swapped in and out of memory to the disk.

Online communication: כל הזמן המחשב מצפה שאכתוב פקודה. ברגע שהמחשב מסיים לבצע פקודה הוא מחכה שוב עד שיקבל פקודה חדשה מהמשתמש.

Personal-Computer Systems:

Computer system dedicated to a single user.

תפקידו היחיד של המחשב הוא לשרת את המשתמש היחיד - לכן פועל על עקרונות של נוחות למשתמש וזמן תגובה מהיר.

They can adopt technology developed for larger OS.

Parallel Systems:

A parallel computer is a computer that has more the one CPU.

Tightly coupled systems: The processors share memory and a clock. Shared memory.

Advantages:
- Increased throughput - תפוקה - כמה תוכניות מספיקים לבצע תוך פרק זמן מסוים.
- אם העבודה תלויה ב-CPU זה חסכוני.
- Increased reliability: graceful degradation and fail-soft systems - אם מעבד אחד הפסיק לפעול - המעבדים האחרים מגבים אותו.

Symmetric Multiprocessing:

Each processor runs an identical copy of the OS. Each processor runs its own kernel.
Many processes can run at once without performance deterioration.
Many problems in scheduling.

A-symmetric Multiprocessing:

Each processor is assigned a specific task. Only one processor runs kernel (master processor) and the others (slaves) don't run it at all. The master computer schedules and allocates work to slave processors.

The communication in parallel system is through the **internal memory** and not through the CPU.

אם זה היה דרך ה-CPU אזי:
1. צריך לתזמן אותם: בדיוק כשמעבד 1 יהיה במצב שליחה שהמעבד השני יהיה במצב מקבל.
2. אם נעצור מעבד אחד ונמשיך את השני ← סיבוך מערכת ההפעלה ובזבוז מעבד.

Distributed Systems:

If there is **one memory** (RAM) then it's a **parallel computer.** If there are **some memories → Distributed**.

Loosely coupled systems: each processor has its own local memory; processor communicate with one another through various communication lines.

Advantages:
- Resource sharing.
- Computation speed up - load sharing.
- Reliability.
- Communication.

Chapter 2:

התקני ק/פ וה-CPU יכולים להתבצע במקביל.
לכל בקר התקן יש buffer פנימי שנמצא בתוך הבקר עצמו.
במערכות ישנות ה-CPU העביר מידע מ/אל הזיכרון הפנימי מ/אל ה-buffer הפנימי. כיום ה-CPU מכיר רק את ה-Cache וה-CPU פוקד על ה-DMA להעביר דברים מהזיכרון הפנימי ל-buffer ולהיפך.
הק/פ מתבצע מההתקן ל-buffer הפנימי של הבקר.
הבקר מודיע ל-CPU שהוא סיים את הפעולה ע"י <u>פסיקה</u>.
<u>מדוע צריך את ה-buffer?</u>
לולא ה-buffer כל בית שאני מקבל מהדיסק כל הזמן היה גורם לפסיקה ← מפריע לעבודת ה-CPU.
לכן נמלא קודם את ה-buffer ורק כשהוא מלא נשלח פסיקה ל-CPU.
לפעמים לא צריך buffer:
למשל, <u>מקלדת</u>: לא הגיוני שנכתוב 1K תווים ורק אז זה יעבור ל-CPU.
<u>מסך.</u>
ה-CPU מבדיל בין התקנים עם buffer והתקנים בלי:
<u>Block Device:</u> התקן שיש לו buffering וה-CPU מדבר איתם בבלוקים שלמים.
<u>Character Device:</u> התקן שאין לו buffering.

הפסיקה היא מספר. כל מספר מייצג סיום של פעולה מסוימת אותה ניתן לזהות לפי מערך הפסיקות שיושב ב-RAM ומכיל את הכתובות של כל הרוטינות שמבוצעות בעת פסיקה מסוימת.
בזמן קבלת הפסיקה מפסיקים תוכנית מסוימת כדי לעבור לביצוע הפסיקה ← לכן, כדי לא לאבד נתונים שקיימים בזמן קבלת הפסיקה, שומרים את הכתובת ומצב ה-CPU של התוכנית שהופסקה כך שניתן יהיה לחזור לתוכנית ולבצע אותה מאותו מקום. מע' ההפעלה שומרת על המצב ע"י אחסון האוגרים וה-program counter.
אם מגיעה פסיקה נוספת בזמן ביצוע פסיקה - נותנים לפסיקה הראשונה לסיים ורק אז מטפלים בפסיקה שהגיעה.
A **<u>trap</u>** is a software generated interrupt caused either by an error or a user request.
מערכת ההפעלה מונעת ע"י פסיקות, כלומר: צריך להריץ את מע' ההפעלה אם יש פסיקה (למשל, בעת פעולת כתיבה/קריאה, חילוק ב-0, גלישה לשטחים של תוכנית אחרת = Segmentation Fault וכו').

<u>DMA Structure</u> (ר' שקף - 2.7)
ה-DMA משמש להוריד לחץ מה-CPU.
בקרי ההתקן מעבירים בלוקים של מידע מה-buffer ישירות לזיכרון הראשי בלי התערבות ה-CPU.
יש פסיקה אחת בלבד לכל בלוק.
משמש להתקני ק/פ מהירים ומאפשר להעביר מידע במהירויות הקרובות למהירות הזיכרון.

<u>Storage Structure:</u>
<u>זיכרון ראשי:</u> מקום אחסון גדול (RAM) שה-CPU ניגש אליו דרך ה-Cache.
<u>זיכרון משני:</u> שלוחה של הזיכרון הראשי שמספקת מקום אחסון לא נדיף.
<u>דיסק מגנטי:</u> יש חלוקה לוגית ל-tracks שמחולקות ל-sectors.

מערכות האחסון מאורגנות לפי הירכיה של: מהירות, מחיר, נדיפות. הנתונים ב-Cache ובזיכרון מתנדפים במהירות בעוד ב-Cdrom, Disk - לא מתנדפים. (שקף - 2.10).
<u>Caching:</u> העתקת מידע למערכת אחסון מהירה יותר.

<u>Hardware Protection:</u>
<u>1. Dual Mode Operation:</u>
הגדרת מצב הביצוע של ה-CPU כ-2 מצבים: 1. user mode, 2. monitor mode.
ב-user mode ה-CPU מוגבל, למשל: אסור לגלוש ממרחב הכתובות המותר לתוכנית.
ב-monitor mode ה-CPU מריץ את ה-kernel מותר לפנות לכל רכיב זיכרון/ I/O. (ה-kernel עצמו רץ במצב של monitor mode).
ב-user mode הביצוע מתבצע מטעם המשתמש ואילו ב-monitor mode הביצוע מתבצע מטעם מערכת ההפעלה.
<u>מימוש:</u> מוסיפים mode bit לחומרת המחשב: ביט = 0 ← monitor mode. ביט = 1 ← user mode.

כל פעם כשיש פסיקה או שגיאה החומרה מעבירה למצב monitor (כיוון שמערכת ההפעלה מונעת ע"י פסיקות). כאשר מע' ההפעלה מסיימת לבצע את מה שהיה צריך לבצע בעקבות הפסיקה היא מעבירה חזרה למצב user (שקף 2.14).
ניתן להריץ privileged insturctions רק במצב monitor.

2. I/O Protection:
כל פעולות ה-I/O הן פקודות פריבילגיות. אף user אינו יכול לפנות ישירות ל-I/O אלא רק דרך מע' ההפעלה, כלומר רק במצב monitor ניתן לפנות ל-I/O.
חייבים להבטיח שאף תוכנית משתמש לא תוכל להשיג שליטה על המחשב במצב monitor.

3. Memory Protection:
חייבים לספק הגנה לזיכרון לפחות עבור מערך הפסיקות והרוטינות שמתבצעות בעת פסיקה. כדי ליצור memory protection מוסיפים 2 אוגרים שקובעים את טווח הכתובות החוקי שתוכנית יכולה לגשת אליהם:

Base Register: holds the smallest legal physical memory address
Limit Register: contains the size of the range. = גודל הטווח = גודל התוכנית
הזיכרון שמחוץ לטווח המוגדר מוגן. אם משתמש מנסה לגשת אל מתחת לתחום - החומרה עוצרת אותו. בעצם ע"י malloc/new רוצים להגדיל את גודל התוכנית וניתן להקצות שטחים לא רצופים (שקף 2.17).
Protection Hardware: (שקף 2.18)
כשפועלים במצב monitor למע' ההפעלה יש גישה לא מוגבלת גם לזיכרון של ה-monitor וגם לזיכרון של ה-user.
פקודות ה-load של אוגרי ה-base וה-limit הן פקודות פריבילגיות (כי אחרת כל אחד יוכל להגדיר לעצמו את טווח הכתובות החוקי ובעצם יוכל לגשת לכל מקום).

4. CPU Protection:
Timer: רכיב תוכנה שסופר כמה clocks עברו. הוא משמש להפסקת המחשב לאחר זמן מוגדר כדי להבטיח שמע' ההפעלה קיבלה שליטה. ה-Timer קטן כל clock tick וכשהוא מגיע ל-0 הוא שולח פסיקה = Time slice exceeded (למשל: תוכנית שיש בה לולאה אינסופית תיפסק בגלל ה-timer). כל פעם כשמריצים תוכנית ה-timer מאותחל.
לרוב משתמשים ב-timer למימוש time sharing. הוא משמש גם לחישוב הזמן הנוכחי.
הפקודה load timer שטוענת את רכיב ה-timer ע"י ה-kernel היא פריבילגית.

פעולות ה-I/O הן פריבילגיות ונעשות רק ע"י ה-kernel. אז איך בכל זאת מבצע המשתמש פעולות I/O? ע"י system calls.
System calls usually take the form of a trap to a specific location in the interrupt vector.
מתבצע מעבר דרך מערך הפסיקות לרוטינה הרצויה במע' ההפעלה וביט ה-mode נקבע למצב monitor.
ה-monitor מוודא שכל הפרמטרים נכונים ומותרים ואז מבצע את הבקשה שבסופה מחזיר את השליטה לתוכנית.

לסיכום, **Hardware Protection** מתבצע ע"י:
1. monitor/user – dual mode operation.
2. I/O protection – פקודות I/O הן פריבילגיות.
3. Memory protection – base & limit register.
4. CPU protection – timer.

Chapter 3:

Process Management:

תהליך היא תוכנית בזמן ביצוע. תהליך דורש מספר משאבים הכוללים זמן CPU, זיכרון, קבצים והתקני ק/פ, עמ"נ לבצע את המשימה שלו.

מע' ההפעלה אחראית על:

ניהול תהליכים:

- יצירת תהליך ומחיקתו.
- הפסקת תהליך והמשכתו.
- אספקת מכניזם לסינכרוניזציה ותקשורת תהליכים.

ניהול זיכרון:

- מעקב אחר חלקי הזיכרון שנמצאים בשימוש וע"י מי הם בשימוש.
- להחליט איזה תהליך לעלות לזיכרון כשמתפנה מקום.
- להקצות/deallocate מקום בזיכרון לפי הנדרש.

ניהול דיסק:

- Free-space management.
- הקצאת מקום אחסון.
- תזמון דיסק.

ניהול קבצים:

- יצירת קובץ ומחיקתו.
- יצירת מחיצה ומחיקתה.
- Support of primitives for manipulating files and directories.
- מיפוי קבצים לאחסון המשני.
- גיבוי קבצים לאמצעי מדיה לא נדיפים.

מערכת ה-I/O מורכבת ממערכת buffer-caching; מנשק כללי device-driver; Drivers להתקני חומרה מסוימים.

Protection Mechanism חייב להבחין בין שימוש מורשה לשימוש לא מורשה; הגדרת הפיקוח שיש לאכוף; לספק אמצעים לאכיפה.

התוכנית שקוראת ומפרשת control statements נקראת shell (או control-card interpreter, command line interpreter). מטרתה היא לקבל ולבצע את הפקודה הבאה לביצוע.

קיימות פונקציות נוספות שמטרתן לא לעזור למשתמש, אלא יותר להבטיח פעולת מערכת יעילה:

- הקצאת משאבים למשתמשים רבים או ל-jobs רבים שרצים באותו זמן.
- Accounting: מעקב ורשימה של אילו משתמשים משתמשים בכמה ובאילו סוגי משאבי מחשב.
- הגנה: להבטיח שכל הגישות למשאבי המע' הן תחת בקרה.

ה-system calls מספקות את המנשק בין תוכנית שרצה ומע' ההפעלה. הן בד"כ זמינות כפקודות אסמבלי.

3 גישות עיקריות משמשות כדי להעביר פרמטרים בין תוכנית שרצה למע' ההפעלה:

1. העברת הפרמטרים באוגרים.
2. אחסנת הפרמטרים בטבלה בזיכרון, כאשר כתובת הטבלה מועברת כפרמטר באוגר.
3. דחיפת הפרמטרים למחסנית ע"י התוכנית והוצאתם מהמחסנית ע"י מע' ההפעלה.

Chapter 4:

מע' ההפעלה מבצעת תוכניות שונות:
Batch systems - **jobs**
Time-Shared systems - **user programs** or **tasks**

job: תוכנית בזמן ריצה. job יכול להכיל כמה תהליכים.
process: הביצוע חייב להתקדם באופן סדרתי - פקודה אחרי פקודה. למשל: אם תוכנית עושה fork() אז היא יוצרת מס' תהליכים. התהליך זה קטע בזיכרון והוא מכיל:
text section: התוכנית עצמה בשפת מכונה.
data section: משתנים גלובליים.
stack section: כל פעם שקוראים לפונק' דוחפים ערכים למחסנית. משתנים מקומיים מוכנסים למחסנית.

יש תהליך ראשון שנוצר בהתחלה (init process) שהוא יכול ליצור תהליך חדש ואז הוא הופך ל"אבא" שלו וכו'.
שיתוף משאבים: יש 3 אפשרויות:

1. Parent and children share all resources.
הגיוני כאשר הם עושים עבודה משותפת - למשל: mergeSort מקבילי בסיבוכיות לינארית.
2. Children share subset of parent's resources.
3. Parent and child share no resources.

ביצוע: 2 אפשרויות:

1. Parent and children execute concurrently.
בייחוד כשיש 2 CPUs. אם יש CPU אחד אז יש עבודה "כאילו" במקביל ע"י שיתוף זמנים.
2. Parent waits until children terminate.

Address Space - שטח: 2 אפשרויות:

1. Child is duplicate of parent - stack, data, text-משכפלים את ה.
2. Child has a program loaded into it - creating new text, data and stack sections.

ברגע שעושים fork() אזי הבן מקבל רק pointer אל האבא, ורק אם משנים משהו בבן אז הוא משתכפל.

תהליך מבצע את הפקודה האחרונה ואז הוא מסיים, או שיש פקודה מפורשת שאומרת לסיים.
האבא מחכה לתוצאת ה-exit() של הבן ובעקבותיה הוא יכול לבצע פעולות מסוימות ומע' ההפעלה משחררת את המשאבים שתפס התהליך שסיים.
כל תהליך יכול לגרום לסיום ביצוע תהליך אחר של אותו משתמש ע"י kill.
כשאבא מסיים: יש הכרזה על הבנים כ"יתומים" ותהליך init מאמץ אותם כבניו.
במע' מסוימות אם תהליך מת הורגים את כל הבנים שלו.

מצבי תהליך (לא ב-Unix):
כשתהליך מתבצע הוא משנה את המצב שלו: (שקף 4.6)
new: התהליך נוצר - כאשר ה-kernel יוצר תהליך, מקצה לו שטחים, מעתיק לו נתונים וכו'.
running: ביצוע פקודות.
התהליך במצב running עד ש:

1. ביקש I/O ואז הוא עובר למצב waiting ולא מועמד להיכנס ל-CPU (רק מי שנמצא במצב ready מועמד להיכנס ל-CPU).
2. time slice exceeded: מחזירים אותו לתוך ה-ready כי הוא עדיין צריך CPU.

ready: the process is waiting to be assigned to a process - כל התהליכים שמוכנים לרוץ.
ה-kernel כל פעם מכניס תהליך אחד מתוך ה-ready ומתחיל להריץ אותו - מצב running.
waiting: התהליך מחכה למאורע מסוים שיקרה.
terminated: התהליך סיים את הביצוע.
כשתהליך מבצע פקודת exit או שנגמרו לו הפקודות הוא עובר למצב terminated ואז יש ביטול הקצאות, הודעה לאבא וכו'.

מצבי תהליך ב-Linux:
R - running/ready
S - sleeping/waiting
T - stopped

Z - zombie - תהליך שמבחינתו יכול לסיים אבל עוד לא הודיע לאבא את ה-exit status (כי האבא במצב wait).

D - uninterruptible sleep (I/O) - תהליך שמחכה ל-device. ההבדל בין זה ל-sleeping הוא שב-D יש עקיפה של מע' ההפעלה ופונית ישירות להתקן. כלומר זהו sleep שלא ניתן להפסיק/להרוג אותו, כי אם נפסיק אותו אז בעצם נפסיק אותו באמצע אינטראקציה עם ההתקן.

<u>Process Control Block (PCB):</u>

ה-kernel שומר על כל תהליך מידע מסוים. הפרטים הטכניים שלו נקראים PCB. המידע שנשמר:

- מצב התהליך.
- program counter - לזכור איפה ה-PC של המצבים שבמצב stop/sleep ושלא רצים ב-CPU כדי שכשנחזור אליהם נדע מאיפה להמשיך.
- אוגרי CPU - שמירת מצבי האוגרים, כדי שכשנטען תהליך מחדש נקבל את ערכי האוגרים שלו.
- מידע המתזמן של ה-CPU: איזה תור, איזו עדיפות וכיו"ב.
- מידע על ניהול הזיכרון: צריך לדעת עבור כל תהליך איפה נמצא כל נתון (איזה חלקים הם ב-RAM ואיזה חלקים בדיסק).
- מידע ניהול: למי שייך התהליך, מה ההרשאות וכו'.
- מידע על מצב I/O - עבור תהליכים שבמצב sleep (תהליך שמחכה שאיזושהי פעולת I/O תתבצע צריך לדעת מה מצב ה-I/O.

ה-ready queue מכיל את כל התהליכים שבזיכרון שזמינים ומחכים לביצוע, כלומר התהליכים שמוכנים לרוץ אבל לא רצים כי תהליך אחר רץ.
ה-device queues מכילים את כל התהליכים שממתינים להתקני I/O.
יש תחלופה של תהליכים בין התורים.

לכל התקן התהליכים נשמרים ברשימה מקושרת כאשר כל תהליך חדש מתווסף לסוף הרשימה בקלות כי לכל התקן יש הצבעה על סוף הרשימה (שקף 4.12).
(שקף 4.13).

<u>המתזמן - שקף 4.14:</u>

המתזמן לזמן ארוך (או מתזמן ה-jobs) בוחר איזה תהליכים יובאו לתוך ה-ready queue (לא קיים ב-UNIX). לא קוראים למתזמן זה הרבה פעמים ולכן הוא יכול להיות איטי וניתן להשתמש בו באלגוריתמים מורכבים יותר. מתזמן זה שולט על דרגת ה-<u>multiprogramming</u>, כלומר: כמה תהליכים רצים במקביל.

המתזמן לזמן קצר (מתזמן CPU) בוחר איזה תהליכים צריכים להתבצע בפעם הבאה ומקצה CPU (כלומר, איזה תהליכים יעברו מה-ready queue ל-CPU). למתזמן זה קוראים די הרבה ולכן הוא חייב להיות <u>מהיר</u>.

<u>ה-dispatcher הוא חלק מה-kernel!</u>

ניתן לתאר תהליכים כ:

<u>I/O bounded process:</u> מבזבז יותר זמן בפעולות I/O מאשר בביצוע חישובים, יש לו short CPU bursts (למשל: בסיסי נתונים).

<u>CPU bounded process:</u> מבזבז יותר זמן בביצוע חישובים. יש לו few very long CPU bursts (למשל: תהליכים מדעיים).

סוגי התהליכים בעצם מעבירים פרמטרים עבור האלגוריתם של המתזמן והם חלק מהשיקולים של המתזמן לזמן ארוך.
כדי שתהיה תמיד ניצולת גם של ה-CPU וגם של ה-I/O המטרה היא שיהיו תהליכים משני הסוגים.

<u>מיתוג הקשר (Context Switch):</u>

כשה-CPU עובר לתהליך אחר המע' חייבת לשמור את המצב של התהליך הישן ולהעלות את המצב השמור עבור התהליך החדש. מיתוג ההקשר הוא שמירת נתוני התהליך הנוכחי ל-PCB ושחזור נתוני התהליך הקודם מ-PCB אחר.

זמן המיתוג הוא overhead, כיוון שהמטרה העיקרית היא הרצת תהליכים, ואילו בזמן המיתוג ה-CPU אינו מנוצל להרצת תהליכים אלא להחלפה של תהליכים.
הזמן הדרוש למיתוג תלוי בחומרה.

<u>תהליכים משותפים:</u>
<u>חסרון תהליך עצמאי:</u> במהלכו לא ניתן להעביר לו מסרים מתהליכים אחרים, כלומר הם לא יכולים להיות מושפעים מתהליכים אחרים.
תהליכים משותפים יכולים להשפיע ולהיות מושפעים מביצוע של תהליך אחר. לתהליך משותף יש קשר לפחות עם תהליך אחד במערכת (לדוגמה: piping).
<u>יתרונות:</u>
- שיתוף מידע: לשני תהליכים יש אותו משתנה והם יכולים לשנות את ערכיו.
- הגברת קצב החישוב: CPU אחד יריץ קטע אחד ו-CPU אחר יריץ קטע שני. תהליך אחד יפנה לבקשת I/O ובאותו זמן התהליך השני ימשיך לרוץ.
- מודולריות.

<u>תקשורת ישירה (signals):</u>

Links are established automatically.
A link is associated with exactly one pair of communicating processes.
Between each pair there exists exactly one link.
The link may be bidirectional, but is usually uni-directional.

למשל: שליחת סיגנלים: התהליך שולח סיגנל בכיוון אחד.
בד"כ לא נהוג לבצע pipe ל-2 כיוונים.

<u>תקשורת לא-ישירה:</u>
יש משאב במערכת (ב-unix: message queue) לתוכו ניתן לשלוח הודעה, וכל תהליך שיגיע אח"כ שרוצה לקבל את ההודעה יוציא את ההודעה מהמשאב.

Messages received from the resource itself. Each resource has a unique id.
Processes can communicate only if they share a resource.
Link established only if processes share a common resource.
A link may be associated with many processes.
Each pair of processes may share several communication links.
Link may be uni-driectional or bi-directional.

<u>בעייה:</u> P1, P2, P3 כולם שותפים במשאב. P1 שולח וגם P2 וגם P3 מקבלים. מי יקבל את ההודעה?
<u>פתרון:</u>
- אפשר ל-link להיות משויך לכל היותר ל-2 תהליכים.
- אפשר רק לתהליך אחד בכל פעם לבצע פעולה שמתקבלת - כלומר, לא ניתן לבצע פעולת receive במקביל אלא רק אחת כל פעם. כלומר, הראשון שלקח הוא זה שיקבל והשני לא. כדי לדאוג שתהליך מסוים יקבל את ההודעה דואגים שהתהליך יגיע ראשון.
- אפשר למע' לבחור באופן שרירותי את המקבל. המע' מיידעת את השולח מי קיבל.

<u>**Buffering:**</u>
ה-message עובר דרך הזיכרון הפנימי ויש כמה דרכים למימוש:
1. <u>**Zero capacity:**</u> לא מקצים למסר מקום בזיכרון (מודל תיאורטי) ← ה-CPU מדברים בינם לבין עצמם ושולחים ביניהם את המסרים.
2. <u>**Bounded capacity:**</u> גודל מוגבל של n בתים. הנתונים נשלחים ואם הצד השני לא מרוקן אותם אזי ה-buffer מתמלא בהם עד שהוא מגיע לגודל המקסימלי ואז נתקע וצריך להמתין עד שהצד השני ירוקן את הנתונים.
3. <u>**Unbounded capacity:**</u> באופן תיאורטי, אין הגבלה על גודל ה-link.

ה-pipe הם מסוג bounded capacity וה-messages הם unbounded.

<u>כיצד ה-links נוצרים?</u>
1. תקשורת ישירה - קשר ברור בין התהליכים.
2. תקשורת עקיפה - נוצר דרך משאב.

<u>האם link יכול להיות מקושר עם יותר משני תהליכים?</u>
1. ה-signals מוגבלים ל-2 תהליכים בלבד.
2. ב-messages יכולים להיות מעורבים 3, 4 תהליכים.

<u>כמה links יכולים להיות בין כל זוג של תהליכים שמתקשרים זה עם זה?</u>
Unix בד"כ לא מגביל.

<u>מה גודל ה-link?</u>
תקשורת בין תהליכים נעשית דרך הזיכרון. כמה מכיל כל קו תקשורת כזה?
signal מכיל 5 ביטים (יש 32 סיגנלים שונים). pipe מכיל גודל קטן (0.5 KB) ו-messages גדול יותר.

<u>האם גודל ההודעה ש-link יכול להכיל קבוע או משתנה?</u>
pipe - גודל קבוע.
messages - גודל משתנה וניתן להגדיל אותם.

<u>האם link הוא דו-כיווני או חד-כיווני?</u>
סיגנלים הם חד כיווניים. אם רוצים עוד סיגנל יוצרים קשר חדש.
messages - דו-כיווניים.

<u>בעיית היצרן-צרכן:</u>

התהליך היצרן יוצר אינפורמציה שנצרכת ע"י התהליך הצרכן. הבעייה: איך ננהל את הקשר ביניהם.

<u>unbounded buffer:</u> אין מגבלה על גודל ה-buffer.
<u>bounded buffer:</u> מניחים שיש גודל buffer קבוע.
השאלה בשני סוגים אלו הם מי ינהל את המגבלות על הגודל: התהליך עצמו או מערכת ההפעלה.
למשל: ב-pipe שהוא bounded מע' ההפעלה בודקת אם לא ניתן להכניס עוד נתונים וכשה-buffer נסתם היא תעצור את התהליך.

<u>פתרונות לבעיית היצרן-צרכן:</u>

<u>Bounded Buffer - Shared Memory Solution: פתרון ע"י זיכרון משותף (שקף 4.24)</u>

❖ ניהול המגבלה ע"י התהליך

```
Shared data
        var n;
        type item = ...;
        var buffer: array[0..n-1] of item;
        var in, out: 0..n-1; { in - לתוכו מכניס היצרן את הנתון, out - מתוכו מוציא הצרכן את הנתונים}
```

<u>תהליך היצרן:</u>
היצרן מייצר איבר במשתנה שנקרא nextp. נתון זה ייצרך לבסוף ע"י הצרכן.
כעת בלולאה מחליט היצרן איפה להכניס את הנתונים ל-buffer. הוא בודק שיש מקום ב-buffer ואם ה-buffer מלא לחלוטין - הוא לא עושה כלום.
<u>תהליך הצרכן:</u> (שקף 4.25)
בלולאה הוא בודק אם יש בכלל מידע לצרוך. אם אין מידע הוא עושה no-op והוא נתקע עד שיהיה מידע.

<u>חסרונות:</u>
- הפתרון נכון, אבל ניתן למלא רק buffer n-1. לא ניתן למלא את כל המערך ותמיד יש איבר אחד ריק כדי להבדיל בין מצב ריק למצב מלא (תמיד יש בדיקה אם out = n+1 שזה מצב מלא, ואילו out = in - מצב ריק).
- no-op do חוזר ומעסיק את ה-CPU כדי לבדוק אם לא לעשות כלום. יש בזבוז של זמן CPU (busy-wait).

<u>**Therads (תהליכון):**</u>
ברגע ש-unix יוצר תהליך חדש ע"י fork הוא מעתיק את ה-text, data ו-stack. מה אם נרצה סה"כ שתהליך חדש ישנה רק משתנה בתוכו, אולם רוצים להשתמש באותו text כמו התהליך הקודם?
אפשר שקטע קוד משותף יהיה משותף לכמה תהליכונים. לתהליכון אין קוד משלו (בניגוד לתהליך) אלא יש לו <u>קוד משותף</u>. גם קטע ה-data משותף בתהליכונים ומשאבים של מע' ההפעלה.
ה-thread הוא יחידה בסיסית של CPU utilization. הוא מורכב מ:
- program counter - כל תהליכון נמצא במקום אחר בתוך הקוד המשותף.

- register set - לכל תהליכון יש אוגרים משלו.
- stack space - אסור שה-stack יהיה משותף.

כל התהליכונים יחד שמבצעים את הקוד המשותף מכונים משימה (task).
בתהליכון קטעי ה-data וה-text משותפים.

ההבדל בין task ל-process הוא שתהליך הוא קטע סדרתי של תוכנית ואילו משימה היא משהו מקבילי.
אם למשימה יש תהליכון אחד - זה שקול למשהו שמתקדם באופן סידרתי = תהליך.

אם תהליכון אחד מבקש I/O אז הוא בעצם נתקע וכל שאר התהליכונים במשימה ממשיכים להתקדם.
שיתוף של כמה תהליכונים באותו job יוצר תפוקה גבוהה וביצועים משופרים.
יישומים שדורשים שיתוף של buffer משותף (למשל: בעיית היצרן-צרכן) נהנים מניצול תהליכונים.
התהליכונים מספקים מכניזם שמאפשר לתהליכים סדרתיים להריץ blocking system calls בעודם מאפשרים להשיג מקביליות.

יש 2 סוגי תהליכונים:
Kernel supported threads: ה-kernel מכיר אותם ונותן להם זמן CPU כשיש צורך.
User-lever threads: ה-kernel לא יודע שיש תהליכון נפרד. המשתמש מחליט איך לחלק את זמני ה-CPU בין התהליכונים שלו בעוד ה-kernel רואה רק תהליך אחד.
ה-hybrid approach מיישמת את 2 סוגי התהליכונים.
LWP – דרגת ביניים בין תהליכוני user level ו-kernel level.

user level thread:
יתרון:
מהירות העברה. ניתן בקלות לעבור בין אחד לשני ולא צריך לעשות את כל ה-context switch בין תהליכון אחד לשני - חסכון בזמן.
חסרונות:
- אם תהליכון אחד ביקש בקשת I/O אז כל ה-task כולו נעצר, כי ה-kernel לא יודע שיש כמה תהליכים שיכולים לרוץ ולכן הוא עוצר את כולם.
- אם יש כמה CPU והמע' לא עמוסה לא ניתן לרוץ על כמה CPU, בעוד הדבר כן אפשרי ב-kernel supported.
- היחס בין התהליכונים: למרות, שלמשל, יש 100 תהליכונים, ה-kernel יתייחס אליהם כאל תהליך אחד ולכן לא ייתן להם יותר זמן CPU מאשר לתהליך שמכיל פחות תהליכונים.

kernel supported thread:
יתרון: אם תהליכון אחד ביקש בקשת I/O. ה-kernel יודע איזה תהליכון ביקש I/O וכל שאר התהליכים יכולים להמשיך לרוץ.
חסרון: עושים **context-switch** מלא.

Resource needs of thread types:

- Kernel thread: small data structure and a stack; thread switching does not require changing memory access information - relatively fast.כל הנתונים כבר ב-kernel
- LWP: PCB with register data, accounting and memory information. Switching between LWPs is relatively slow. כדי לעבור לתהליך צריך להעלות נתונים מה-PCB
- User-level thread: only need stack and a program counter; no kernel involvement means fast switching. Kernel only sees the LWPs that support user-level thread.

(שקף 4.30)

Chapter 5:

Maximum CPU utilization obtained with multiprogramming.

פרצי עיבוד של CPU: ביצוע תהליך מורכב ממחזורים של ביצוע CPU והמתנות I/O.
(שקף 5.3)

<u>מתזמן CPU:</u>
בוחר מבין התהליכים בזיכרון שמוכנים לרוץ, ומקצה את ה-CPU לאחד מהם.
<u>מתי מפעילים את המתזמן?</u>

1. בקשת I/O - מעבר ממצב running ל-waiting.
2. time slice exceeded - מעבר מ-running ל-ready.
3. כשמקבלים תשובה מה-I/O יש פסיקה ועוברים ל-kernel שמטפל בבקשת ה-I/O - מעבר מ-waiting ל-ready.
4. כשתהליך מסתיים - מעבר ל-terminate.

התזמון ב-1 וב-4 הוא **nonpreemptive** ואילו 2 ו-3 הם **preemptive**, כלומר "לוקחים בכוח". כאשר תהליך מסוים רץ על ה-CPU הרבה זמן ואנו רוצים להפסיק אותו או כשמגיעה פסיקה אז "לוקחים לו בכוח" את זמן ה-CPU ומעבירים את ה-CPU ל-kernel.

<u>ה-**Dispatcher**:</u> תפקידו - להתחיל להריץ תהליך של המשתמש.
מודול ה-dispatcher נותן שליטה על ה-CPU לתהליכים שנבחרו ע"י המתזמן לזמן קצר. זה דורש:

- מיתוג הקשר.
- מעבר למצב user.
- קפיצה למיקום המדויק בתוכנית המשתמש כדי להתחיל מחדש את התוכנית (קפיצה לנקודה שממנה הפסקנו כשעזבנו את התהליך בפעם הקודמת).

Dispatch latency: הזמן שלוקח ל-dispatcher לעצור תהליך אחד ולהתחיל תהליך אחר.

<u>קריטריוני התזמון:</u>

- <u>ניצולת CPU:</u> לשמור על ה-CPU עסוק כמעט כל הזמן.
- <u>תפוקה:</u> מספר התהליכים שמסיימים את הביצוע שלהם בכל יחידת זמן (לפי Benchmark). התפוקה יכולה להיות תלויה גם במידה <u>החפיפה</u> בין התהליכים.
- <u>Turnaround time (elapsed time/execution time):</u> הזמן לביצוע תהליך <u>אחד</u> מסוים.
- <u>זמן המתנה:</u> זמן שתהליך חיכה ב-ready queue.
- <u>זמן תגובה:</u> כמה זמן לקח מאז שהתחלנו להריץ את התוכנית ועד שקיבלנו את התגובה הראשונה (לאו דווקא כל ה-output).

<u>קריטריונים לאופטימיזציה:</u>

- מקסימום ניצולת CPU.
- מקסימום תפוקה.
- מינימום turnaround time.
- מינימום זמן המתנה.
- מינימום זמן תגובה.

<u>שיטות תזמון:</u>

<u>1. First Come, First Served (FCFS) (שקפים 5.8, 5.9)</u>
בשיטה יכול להיווצר אפקט השיירה **(convoy effect)** - short process behind long process.

<u>2. Shortest Job First (SJF) (שקפים 5.11, 5.12)</u>
לשייך לכל תהליך את אורך פרץ העיבוד הבא שלו ולהשתמש באורכים אלו כדי לתזמן את התהליכים עם הזמנים הקצרים ביותר. ניתן ליישם ב-2 דרכים:

- **nonpreemptive**: ברגע שה-CPU ניתן לתהליך לא ניתן לתת אותו לתהליך אחר עד שמסתיים פרץ העיבוד. (שקף 5.11)
- **preemptive**: אם מגיע תהליך עם אורך פרץ CPU קטן מהזמן הנותר לתהליך שמבוצע כעת אז נותנים לו את זמן ה-CPU (Shortest Remaining Time First - SRTF). שיטה זו היא אופטימלית - נותנת מינימום זמן המתנה ממוצע לקבוצת תהליכים נתונה. (שקף 5.12)

<u>קביעת אורך פרץ ה-CPU הבא:</u>

ניתן רק להעריך את האורך ע"י הערכה אקספוננציאלית בהסתמך על פרץ ה-CPU הקודם.
t_n = actual length of nth CPU burst. זמן פרץ העיבוד ה-n
τ_{n+1} = predicted value for the next CPU burst. כמה מעריכים שיהיה פרץ העיבוד הבא
$0 \le \alpha \le 1$ - מס' האחוזים שניתנו

ונגדיר: $\tau_{n+1} = \alpha t_n + (1-\alpha)\tau_n$

<u>הסבר:</u> α מייצג את מס' האחוזים שניתנו. לכן כופלים את α באורך פרץ ה-CPU שהיה בפועל ואת האחוזים הנותרים (1-α) בכמה שהערכנו שיהיה.

<u>למשל:</u>
כאשר α = 0 אזי $\tau_{n+1} = \tau_n$. כלומר, לא מתחשבים בהיסטוריה האחרונה.
כאשר α = 1 אזי $\tau_{n+1} = t_n$. כלומר, רק פרץ ה-CPU האחרון נחשב.

אם נפשט את הנוסחה נקבל: $\tau_{n+1} = \alpha t_n + (1-\alpha)\alpha t_{n-1} + \ldots + (1-\alpha)^j \alpha t_{n-j} + \ldots + (1-\alpha)^{n+1}\tau_0$
וניתן לראות שכיוון ש-α ו-1-α הם קטנים או שווים ל-1 אז לביטוי הראשון יש הכי הרבה משקל, כלומר מתייחסים הכי ברצינות לפרץ העיבוד האחרון, וככל שפרץ העיבוד הוא יותר עתיק בהיסטוריה - נותנים לו פחות משקל.

<u>3. Priority Scheduling:</u>
לכל תהליך מוצמד מספר עדיפות. ה-CPU מוקצה לתהליך עם העדיפות הגבוהה ביותר (כאשר ככל שהמספר קטן יותר ← עדיפות גבוהה יותר). מבוטא ב-unix לפי nice, כאשר ככל שה-nice יותר גבוה התהליך מקבל פחות CPU. ניתן ליישם כ-(non)preemptive.
SJF הוא סוג של תזמון עדיפויות כאשר העדיפות נקבעת לפי פרץ העיבוד המשוער הבא (זמן CPU חזוי ארוך יותר ← עדיפות נמוכה יותר).

<u>בעיה: Stravation</u> - יש סיכוי שתהליכים שהעדיפות שלהם נמוכה לעולם לא יתבצעו.
<u>פתרון: Aging:</u> עם הזמן מעלים את העדיפות של התהליך.

<u>4. Round Robin: (שקף 5.17, 5.19)</u>
לכל תהליך נותנים פלח זמן (time quantum) שהוא בד"כ נע בין עשרות מילי-שניות ואחרי שהזמן הזה חולף עוצרים את התהליך (preemptive) ומביאים תהליך אחר. את התהליך שהוצאנו מכניסים לסוף התור וכן הלאה - עושים סבב בין התהליכים.
אם יש n תהליכים ב-ready queue ופלח הזמן הוא q, אזי כל תהליך מקבל 1/n מזמן ה-CPU כל פעם בפלחים של עד q יחידות זמן. מזה יוצא שאף תהליך לא מחכה יותר מ-(n-1)q יחידות זמן.
שיטה זו מתנהגת כמו FCFS (FIFO) כאשר q גדול.
כאשר q קטן יש המון context switch לכן יש לדאוג ש-q יהיה גדול ביחס למיתוג ההקשר, אחרת ה-overhead יהיה גדול מאוד.
הצעה: לבחור q כך ש-80% מהתהליכים יספיקו לסיים את פרצי העיבוד שלהם תוך פלח הזמן הזה.

<u>המשפט הבא אינו נכון:</u> אם נבחר <u>q קטן</u> יותר אזי ה-turnaround time יהיה <u>נמוך יותר</u>.

<u>**Multilevel Queue:** (שקף 5.21)</u> תור רב-רמות. יש מס' תורים שממתינים ל-CPU וצריך לדעת לאיזה תור להכניס.
ה-ready queue מחולק ל-2 תורים: foreground (interactive) שמתוזמן לפי Round Robin ו-backgounrd(batch) שמתוזמן לפי FCFS.
יש לבצע תזמון בין התורים. 2 שיטות:

- **Fixed priority scheduling:** לשרת את כל אלו מהחזית ואח"כ את אלו מהרקע. אפשרות להרעבה.
- **Time slice:** כל תור מקבל כמות מסוימת של זמן CPU שהוא יכול לתזמן בין התהליכים שמצויים בו, למשל: 80% to foreground in RR, 20% to background in FCFS.

<u>Multilevel Feedback Queue (שקף 5.23, 5.24):</u> תזוזה בין התורים.
ניתן להעביר תהליך בין התורים השונים, וכך בעצם ניתן ליישם Aging.
מתזמן של multilevel feedback queue מוגדר לפי:

- מס' התורים.
- אלגוריתם תזמון לכל תור.

- שיטה המאפשרת להחליט מתי לשדרג תהליך.
- שיטה המאפשרת להחליט מתי "להוריד בדרגה" תהליך.
- שיטה המאפשרת להחליט לאיזה תור ייכנס תהליך כאשר התהליך צריך שירות.

כל התהליכונים מתחילים באותה רמת עדיפות.
כאשר ה-time slice exceeds לתהליכון מסוים הוא עובר לסוף התור הנמוך שמתחתיו.
ה-time quanta גדל יותר ויותר ככל שהעדיפות יורדת, כך שבסוף הוא מגיע לאינסוף (למשל: אין time slicing ברמה הנמוכה ביותר).
היתרון: נותן טיפול מועדף ל-jobs קצרים על חשבון jobs ארוכים יותר.

Chapter 6:
(שקף 6.1)

<u>בעיית הקטע הקריטי:</u>
אסור לעצור תהליך באמצע שכן אם נעצור אותו באמצע נקבל תוצאות שונות.

נרצה כעת לפתור את בעיית היצרן-צרכן שבה השתמשנו רק ב-n-1 איברים ב-buffer.

(שקף 6.3, 6.4) נשנה את הקוד המופיע בשקף 4.24 (עמ' 8) ע"י הוספת משתנה counter שיאותחל ב-0 ויקודם כל פעם שמשתנה חדש יתווסף ל-buffer. כאשר counter = n נגיד ליצרן לעצור. כאשר counter = 0 הצרכן מפסיק לעבוד כי ה-buffer ריק.

```
Shared data
        type item = ...;
        var bufer: array [0..n-1] of item;
        var in, out : 0..n-1;
        var counter: 0..n; {מונה למנות כמה איברים יש ב-buffer}
        in, out, counter := 0;
```

כיוון שהמשתנים יושבים בד"כ בזיכרון ולא באוגרים יש בעיה, שאולי כאשר נעבור בין התהליכים נקבל תוצאות לא רצויות, לכן פעולות הקידום והורדת ערך המשתנה counter מוגדרות כפעולות <u>אטומיות</u>.

<u>בעיית הקטע הקריטי:</u>
ישנם n תהליכים שכולם רוצים לגשת למידע משותף, אולם אנו לא רוצים שייגשו אליו יחד.
לכל תהליך יש קטע קוד שנקרא קטע קריטי שרק בו ניתן לגשת למידע המשותף. לפני הקטע הקריטי יש הצהרה שמעכשיו אסור להפריע לתהליך, ובסוף הקטע יש הצהרה שהתהליך סיים לבצע את הקטע הקריטי שלו.
<u>הבעיה:</u> להבטיח שכאשר תהליך אחד מתבצע בקטע הקריטי שלו, אף תהליך אחר לא מורשה לבצע את התהליך הקריטי שלו.

<u>פתרון לבעיית הקטע הקריטי חייב לקיים את 3 התנאים הנ"ל:</u>
1. מניעה הדדית: אם תהליך מסוים מתבצע כעת בקטע הקריטי שלו, אז לתהליכים האחרים אסור לבצע את הקטע הקריטי שלהם. כלומר, לא יכולים להיות 2 תהליכים בעת ובעונה אחת בקטע הקריטי. אם תהליך נמצא בקטע הקריטי הוא מונע מתהליכים אחרים להיכנס לקטע הקריטי.
2. התקדמות: אם אף תהליך לא מתבצע כעת בקטע הקריטי שלו ויש כמה תהליכים שרוצים להיכנס לקטע הקריטי שלהם, אזי בחירת התהליכים שייכנסו לקטע הקריטי לא יכולה להידחות ללא סוף (no deadlock!). כלומר, שלא ייווצר מצב שיש תהליכים שרוצים להיכנס לקטע הקריטי ואף אחד מהם לא נכנס אליו. אם יש תהליך שרוצה להיכנס לקטע הקריטי ואף אחד לא בתוכו ← ניתן לו להיכנס.
3. המתנה חסומה: חייב להיות חסם על מס' הפעמים שתהליכים אחרים רשאים להיכנס לקטע הקריטי שלהם לאחר שתהליך מסוים ביקש להיכנס לקטע הקריטי שלו ולפני שאושר לו להיכנס. כלומר, מס' התהליכים שייכנסו <u>לפני</u> התהליך לקטע הקריטי הוא מס' סופי.

<u>הפתרונות הבאים אינם עונים לבעיה:</u>
- אף תהליך לא נכנס אף פעם לקטע הקריטי. פתרון זה עונה רק לתנאי ה<u>מניעה ההדדית</u>.
- תהליך שנכנס ויוצא כל הזמן מהקטע הקריטי עומד ב<u>מניעה הדדית</u> ו<u>התקדמות</u>, אולם שאר התהליכים שרוצים להיכנס יקופחו.

<u>ניסיונות לפתור את הבעיה:</u>
נתונים 2 תהליכים: P0, P1.
- <u>אלגוריתם 1:</u>

```
shared variable:
        var turn: (0..1); {של מי התור}
        initially turn = 0
```

כאשר turn = i אז Pi יכול להיכנס לקטע הקריטי.
כל Pi מבצע:

```
repeat
```

while turn ≠ i do no-op; {כל עוד התור לא שלי - תמתין}
critical section
turn := j; {בסיום הקטע הקריטי התור עובר לשני}
reminder section
until false

הפתרון עונה למניעה הדדית והמתנה חסומה אבל לא להתקדמות, כי למשל אם תהליך j רוצה להיכנס 5 פעמים ו-i רוצה להיכנס 100 פעמים, אזי אחרי 5 פעמים תהליך j לא נכנס יותר לקטע הקריטי ולכן turn = I, אולם כאשר תהליך i ייכנס לקטע הקריטי אזי בסוף turn = j והוא יישאר ככה!

- **אלגוריתם 2:**

shared variables:
var flag: array [0..1] of boolean;
initially flag[0] = flag[1] = false.

כאשר flag[i] = true אזי Pi מוכן להיכנס לקטע הקריטי. כל Pi מבצע:

repeat
flag[i] := true;
→ while flag[j] do no-op; {אם יש מישהו אחר בקטע הקריטי - ממתינים}
critical section
flag[i] := false;
reminder section
until false

הפתרון עונה למניעה הדדית והמתנה חסומה אבל לא להתקדמות, כי אם, למשל, נפסיק את התהליך במיקום החץ. אם ניקח לו בדיוק שם את ה-CPU אזי הדגל שלו יודלק, התהליך יופסק, נעבור לתהליך j. בתהליך j נדליק גם כן את הדגל אולם כעת תהליך j בודק האם תהליך i נמצא בקטע הקריטי וזאת לפי הדגל של תהליך i. אולם הדגל של תהליך i דלוק למרות שהוא אינו בקטע הקריטי ולכן תהליך j לא יוכל להמשיך, ואחרי שיעבור זמן ה-CPU נחזור לתהליך i אולם הוא לא יוכל להמשיך כי הדגל של j דלוק ← קיפאון - **deadlock** ← אין התקדמות.

- **אלגוריתם 3: עונה על כל הקריטריונים:**

כל Pi מבצע:

repeat
flag[i] := true; {הכרזה שאני רוצה להיכנס לקטע הקריטי}
turn := j; {קודם כל נתונים את התור לתהליך השני}
while (flag[j] and turn = j) do no-op; {בודקים האם תהליך j נמצא כבר בקטע הקריטי ואם התור שלו}
critical section
flag[i] := false;
reminder section
until false

אין בעיה כיוון שלמרות שיכול להיות שהדגלים של שניהם יהיו true, ל-turn יש רק ערך אחד: או i או j. flag[j] הוא true רק אם תהליך j רוצה להיכנס לקטע הקריטי והוא לא מקבל ערך אוטומטי בסוף תהליך i.

אלגוריתם Bakery: פתרון הקטע הקריטי עבור n תהליכים:

לפני הכניסה לקטע הקריטי כל תהליך מקבל מספר. התהליך עם המספר הנמוך ביותר מורשה להיכנס לקטע הקריטי.
אם שני תהליכים, Pi and Pj, מקבלים אותו מספר אז אם i < j אז Pi נכנס קודם, אחרת Pj ייכנס קודם.
(אם יש אותו מספר נשבור את השוויון לפי ה-pid שהוא יחיד לכל תהליך כי הוא ניתן ע"י ה-kernel שמחלק את מס' ה-pid בסדר עולה).
יצירת המספרים היא תמיד בסדר לא יורד.

הגדרות:

(a,b) < (c,d) if a<c or if a=c and b<d. (ticket#, pid #).
max(a0, ..., an-1) - מחזיר את המספר הגדול ביותר מבין ה-a

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

כל מבני הנתונים מאותחלים ל-0 ול-false.

```
repeat
    choosing[i] := true; {התהליך בוחר עכשיו מספר}
    number[i] := max(number[0], ..., number[n-1]) + 1; { בוחרים את המס' האחרון שקיים ומוסיפים לו 1}
    choosing[i] := false; {התהליך סיים לבחור מספר }
    for j:=0 to n-1 {עוברים על כל התהליכים במע', כלומר על כל ה-pid שבמע'}
            do begin
  /* אם תהליך אחר באמצע בחירת מס' מחכים שהוא יסיים ורק אז משווים איתו */
  /* ממתינים כי יכול להיות שהתהליך יבחר pid כמו שלי ואז כיוון שה-pid שלי נמוך יותר אזי לו תהיה עדיפות */
  /* יש אפשרות לבחור אותו מס' כיוון שתהליך יכול להיעצר באמצע חישוב פונקציית ה-max ואז יש אפשרות */
  /* שיהיה אותו מס' לכמה תהליכים. */
                while choosing[j] do no-op;
               /*אם number = 0 אזי התהליך לא מעוניין להיכנס לקטע הקריטי ואין טעם להשוות איתו */
                while number[j] != 0 and (number[j],j) < (number[i], i) do no-op;
            end;
        critical section
    number[i] := 0;
        reminder section
until false;
```

<u>הבעיה</u> של האלגוריתם היא זמן.

<u>Synchronization Hardware:</u>

הבעיה הראשונית הייתה שפעולה התבצעה ועצרנו אותה באמצע. הפתרון הוא הגדרה בחומרה כך שאי-אפשר יהיה להפסיק <u>פעולות אטומיות</u> באמצע וזה יתן פתרון לבעיית הקטע הקריטי.

למשל: נרצה לבדוק ולשנות תוכן של מילה באופן אטומי:

```
function Test-and-Set(var target: boolean): boolean;
  begin
        Test-and-Set := target;
        target := true;
  end;
```

<u>מניעה הדדית:</u>

Shared data: var lock: boolean (* initially false *)

תהליך Pi:

```
repeat
    while Test-and-Set(lock) do no-op;
  /* התהליך הראשון שמנסה להיכנס לקטע הקריטי מבצע את הפונקציה Test-and-Set שמקבלת את */
  /* הערך false ולכן הוא ייכנס לקטע הקריטי. כעת ה-lock הוא true ולכן אם תהליך נוסף ינסה */
  /* להיכנס לקטע הקריטי הוא תחילה יפעיל את הפונקציה ואז יקבל true וימתין. */
            critical section
    lock := false;
            reminder section
until false;
```

<u>הבעיה:</u> אין בטחון שלא יהיה תהליך שיהיה מורעב. אין חסם למס' הפעמים שתהליך יכול לחכות.
<u>השימוש ב-t&s מבטיח התקדמות ומניעה הדדית אבל לא המתנה חסומה.</u>

<u>הפתרון: **Semaphore**</u> עוזר לסנכרון בין תהליכים.

ה-Semaphore מכיל מס' שלם. כאשר ה-semaphore קטן או שווה ל-0 ← מישהו נמצא בקטע הקריטי ואסור להיכנס אליו.

ה-semaphore לא דורש busy-waiting.

<u>Semaphore can only be accessed via 2 atomic operations.</u>

בהתחלה ה-semaphore מאותחל ב-1.

```
/* wait(S) מתבצע בכניסה לקטע הקריטי. כל מי שמנסה להיכנס לקטע הקריטי מוריד את ערך ה-semaphore ב-1 */
wait(S):        while S <= 0 do no-op; {בעיית busy-wait - יש בזבוז זמן כל פעם לבדוק אם S<=0}
                        S := S - 1;
                                        /* כשיוצאים מהקטע הקריטי מעלים את S ב-1. */
signal(S)       S := S + 1;
```

הפעולות הנ"ל הן אטומיות, לכן לא ניתן להפסיק אף אחת מהן באמצע וכך לא יכול להיווצר deadlock.

<u>דוגמה:</u> קטע קריטי עבור n תהליכים:

```
Shared variables
        var mutex: semaphore { initially mutex = 1 }
```

כל Pi:

```
repeat
        wait(mutx); {entry section }
                critical section
        signal(mutex); { exit section }
                reminder section
until false;
```

<u>מימוש Semaphore:</u>

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

מגדירים תור של כל התהליכים ← מניעת הבעייה של <u>המתנה חסומה</u>. כשמישהו יצא מהקטע הקריטי אזי הראשון בתור ייכנס אליו. כל תהליך לא מחכה יותר ממס' התהליכים שחיכו לפניו להיכנס.

מניחים 2 פעולות:

block משעה את התהליך - קריאה ל-kernel: התהליך אינו מעוניין יותר לקבל זמן CPU עד שמישהו יעשה לתהליך wakeup. ברגע שתהליך מבצע block הוא מוצא מתור התהליכים המוכנים לרוץ. התהליך עצמו מבצע block.

wakeup(P) ממשיך את הביצוע של תהליך P שהוא blocked = החזרת התהליך לתור. מישהו אחר עושה wakeup תוך ציון התהליך שהוא רוצה להחזיר לביצוע.

וכעת נגדיר:

```
wait(S):        S.value := S.value - 1; { הורדת ערך הסמפור ב-1. אם הערך הוא 0 אז נכנסים לקטע הקריטי}
                if S.value < 0
                then begin
                        add this process to S.L; {הוספת התהליך לתור}
                        block; {התהליך מפסיק לקבל זמן CPU עד שמישהו יעיר אותו}
   /* ה-block מבוצע כדי למנוע בעיית busy-wait . יעירו אותו רק כאשר תהליך אחר יצא מהקטע הקריטי */
                                                                      /* ולכן תהליך זה יוכל להתבצע */
                end;

signal(S):      S.value := S.value + 1;
                if S.value <= 0 { if the condition is true → there are processes in the queue and we need }
                then begin      {  to wake them                                                          }
                        remove a process P from S.L; {לוקחים את הראשון בתור ומעירים אותו}
                        wakeup(P);
                end;
```

<u>**שימושים ב-Semaphore: Semaphore as General Synchronization Tool:**</u>

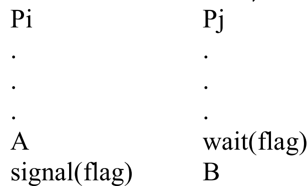
ביצוע B ב-Pj רק לאחר ביצוע A ב-Pi.

הבעיה: את A מריץ תהליך אחד Pi ואילו את B מריץ תהליך אחר Pj.

משתמשים ב-semaphore flag שמאותחל ל-0 ← כאילו יש מישהו בתוך הקטע הקריטי, אבל אין קטע קריטי באמת:

אם A מתבצע קודם ואז B אז אין בעיה וה-semaphores לא מפריעים: A מתבצע ואז מבצע signal ← ה-semaphore עכשיו שווה ל-1, B עושה wait ← ה-semaphore קטן ל-0 ו-B מתבצע.

אם B מתבצע קודם אז B עושה wait. כעת ה-semaphore הוא -1 ולכן B עושה block. B ממתין עד שמתבצע signal. ה-signal מתבצע רק לאחר ביצוע A ורק אז B יוכל לרוץ:

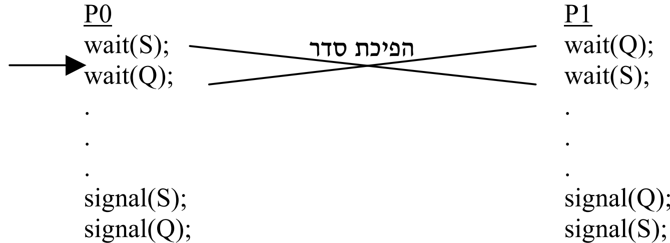


:Deadlock and Starvation

Deadlock: 2 or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes.

אם יש זכרונות משותפים אז לכל שטח זיכרון מקצים semaphore משלו. נניח שיש קטע קריטי עבור 2 שטחי זיכרון ← צריך לעשות wait ל-2 semaphores.

בעיה: נניח ש-S, Q הם 2 semaphors שמאותחלים ל-1. עושים wait בהצלבה ← יכול להיווצר deadlock.



הבעיה כאן שאם נעצור איפה שהחץ אז נבצע רק wait(S) ← S = 0 ואנחנו בתוך הקטע הקריטי מבחינת S. עכשיו עוברים לתהליך P1. עושים wait(Q) ← Q = 0. כעת P1 מנסה לעשות wait(S) ← S = -1 ולכן הוא עושה block. חוזרים ל-P0 שעושה עכשיו wait(Q) ← Q = -1 ולכן גם הוא עושה block.
2 התהליכים נמצאים במצב של בלוק ולא נכנסים לקטע הקריטי ← deadlock.

Starvation: indefinite blocking. A process may never be removed from the semaphore queue in which it is suspended. There is no way to know how many processes will get the CPU before a starved process.

starvation ≠ deadlock. ב-starvation רק אחד לא מצליח להיכנס לקטע הקריטי.

יש 2 סוגי semaphores:

1. **counting semaphore**: אין הגבלה על הערך של ה-semaphore, כאשר מס' שלילי מייצג שכמה תהליכים מחכים בתור. מס' חיובי אפשרי כאשר מאפשרים עד ל-n תהליכים להיות באותו קטע קוד מסוים, כאשר כל תהליך שנכנס לקטע הקוד מוריד את הערך ב-1.
2. **binary semaphore**: הערך יכול לנוע רק בין 0 (= אין מישהו בקטע הקריטי) ל-1 (= יש מישהו בקטע הקריטי). יותר פשוט ליישום.

:Implementing S as a binary semaphore

```
Data structures:
        var     S1: binary-semaphore {סמפור זמני להגנת C}
                S2: binary-semaphore {הסמפור הכללי לקטע הקריטי הגדול}
                C: integer
Initialization: S1 = 1
                S2 = 0
                C = initial value of semaphore S
```

למשל: C = 1 אם רוצים לפתור את בעיית הקטע הקריטי.
C = 4 אם מאפשרים ל-4 תהליכים להיות יחד בקטע קוד מסוים.

```
wait:           wait(S1);
                C := C - 1;
                if C < 0
                then begin
                        signal(S1);
                        wait(S2);
                 {S2 is initialized to 0 so the wait is actually creating block - locking the process }
                   end
                else signal(S1);
```

הפעולות האלו אינן אטומיות, אבל אסור להפסיקן באמצע, לכן יש סמפור S1 שמגן עליהן. בעצם השימוש במשתנה C הוא קטע קריטי

```
signal:         wait(S1); {כדי להגן על C}
                C := C + 1;
                if C <= 0 then signal(S2);
```

/* אם C<=0 ז"א שיש את מי להעיר ולכן עושים signal(S1) ומעירים את התהליך שעשה wait(S2) ונכנס ל-block. */

signal(S1);

בעיות קלאסיות של סינכרוניזציה (בעיית היצרן-צרכן):

הערה: פתרון באמצעות semaphores עשוי לגרום לבעיות starvation ו-deadlock.

1. Bounded Buffer Problem: (שקף 6.26-6.28)

```
Shared data
    type    item = ...
    var     buffer = ...
            full, empty, mutex : semaphore;
            nextp, nextc : item;
            full := 0; empty := n; mutex := 1; {initializing the semaphores}
```

Producer process:

```
repeat
            ...
    produce an item in nextp
            ...
    wait(empty); {empty:= empty-1.{היצרן מוריד אותו בפעם הראשונה ל-n-1 כי הם פונים לאותו קטע זיכרון}
    wait(mutex);
            /* ה-mutex בא לפתור את בעיית הקטע הקריטי הרגילה, ולכן גם האתחול שלו הוא רגיל (ערך 1) */
            ...
    add nextp to buffer
            ...
    signal(mutex);
    signal(full); {כל פעם שהיצרן יוצר איבר בתוך ה-buffer ה-full עולה ב-1}
until false;
```

<u>הסבר:</u> אם הצרכן לא רץ אז היצרן יבצע כל הזמן wait(empty) עד ש-empty = 0, ואז, בפעם ה-n+1, הוא יהיה שלילי ולכן ייתקע ← אין יותר מקום ב-buffer, כל ה-buffer מלא ולכן הוא נתקע.

Consumer process:

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

הצרכן עושה הפוך. כל צריכת איבר הוא מוריד את ערך full ב-1 והוא נעצר כאשר ה-buffer ריק, כלומר כאשר full = 0 - הוא מנסה להוריד אותו ל-1- וממתין.

2. Readers-Writers Problem: (שקף 6.29, 6.30)

<u>הבעיה:</u> יש מידע משותף, מס' תהליכים שרוצים לכתוב ל-buffer המשותף ומס' תהליכים שרוצים לקרוא מה-buffer המשותף. ההבדל מבעיית הקטע הקריטי: אם רק תהליכים קוראים רוצים לגשת יחד לקטע הקריטי אזי זה <u>מותר</u>.

```
Shared data
    var     mutex, wrt: semaphore; {initial value of both is 1}
            readcount: integer; {initial value of readcount is 0}
```

Writer process

```
    wait(wrt); {לפני פעולת הכתיבה}
                ...                          }
            writing is performed             }  כמו בבעיית הקטע הקריטי
```

```
        ...
signal(wrt); {אחרי פעולת הכתיבה}
```

Reader process

```
wait(mutex); {readcount הגנה על המשתנה}
/* mutex מגן על הקטע הקריטי בתוך הכניסה לקטע הקריטי עצמו */
        readcount := readcount + 1;
        /* readcount מייצג כמה תהליכים קוראים נמצאים בתוך הקטע הקריטי (יכול להיות יותר מ-1) */
        if readcount = 1 then wait(wrt);
/* הראשון שנכנס לקטע הקריטי לשם קריאה גורר ל-readcount להיות שווה ל-1 ואז הוא עושה wait(wrt) = הורדת סמפור wrt ל-0 ← מניעה מתהליכים כותבים להיכנס */
signal(mutex);
            ...
        reading is performed
            ...
wait(mutex);
        readcount := readcount - 1;
        if readcount = 0 then signal(wrt);
/* אם אני התהליך הקורא האחרון אז readcount = 0 ועושים signal(wrt) ומאפשרים לתהליכי כתיבה להיכנס לקטע */
signal(mutex);
```

נוצרת אפשרות של **starvation** לתהליכי הכתיבה.

3. Dining-Philosophers Problem:

הבעייה: יש 5 פילוסופים שיושבים סביב שולחן. כל פילוסוף או חושב או אוכל. אם הוא חושב הוא לא צריך משאבים. אם הוא אוכל הוא צריך את 2 המזלגות: משמאלו ומימינו. כיוון שיש משאבים משותפים (=המזלגות) ← בעיית קטע קריטי.

פתרון: כל פילוסוף כשהוא רעב מרים מזלג ימני, אם אפשר. אם הוא תפס את המזלג הוא מנסה לתפוס את המזלג השמאלי. אם יש לו 2 מזלגות אזי הוא מתחיל לאכול.
הבעיה: אם כולם מרימים את המזלג הימני אז כל אחד יחכה שהשמאלי יתפנה ← deadlock.

```
Shared data
        var chopstick: array [0..4] of semaphore; {initial value of all semaphores is 1}

Philosopher i:
repeat
        wait(chopstick[i]); {הרמת המזלג מימין}
        wait(chopstickp[i+1 mod 5]); {ניסיון להרים את המקל משמאל. mod 5 כי הם יושבים במעגל}
                ...
                eat
                ...
        signal(chopstick[i]); {שחרור מזלג ימין}
        signal(chopstick[i+1 mod 5]); {שחרור מזלג שמאל}
                ...
                think
                ...
until false;
```

Chapter 7:

<u>בעיית ה-deadlock:</u>

יש כמה תהליכים שתקועים. תהליך אחד שמחזיק משאב אחד וממתין למשאב אחר, והתהליך השני מחזיק משאב שהשני ממתין לו וממתין למשאב שהראשון מחזיק.

דוגמה:

```
P0              P1
wait(A);        wait(B);
wait(B);        wait(A);
```

ניקח את זמן ה-CPU במיקום החץ. זאת אומרת לאחר שגרמנו לכך ש-A = 0. ב-P1 אנו מורידים את הערך של B ל-0, מנסים לעשות wait(A) ולא מצליחים כי A = 0, לכן חוזרים ל-P0 אולם B = 0 לכן גם ה-wait(B) נתקע.

<u>דוגמה נוספת: בעיית חציית הגשר:</u>

התנועה בגשר הצר היא בכיוון אחד בלבד. ניתן להסתכל על כל קטע של הגשר כמשאב. אם מגיעים לקיפאון ניתן לפתור אותו אם מכונית אחת תזוז אחורנית (preempt resources and rollback) - כלומר, חזרה למצב לפני הכניסה לקטע הקריטי (← ביטול כל הפעולות שהתהליך עשה בתוך הקטע הקריטי). יכול להיות שנצטרך להזיז כמה מכוניות אחורה לאחר שנוצר קיפאון. יכול להיווצר מצב של <u>הרעבה</u>.

<u>Deadlock יכול להיווצר אם 4 התנאים הבאים קורים בו-זמנית:</u>

1. מניעה הדדית: כאשר רק תהליך אחד כל פעם יכול להשתמש במשאב.
2. מחזיק וממתין **(hold and wait)**: תהליך שמחזיק לפחות משאב אחד ממתין לקבל עוד משאבים שמוחזקים ע"י תהליכים אחרים.
3. **No preemption**: משאב יכול להשתחרר רק ביוזמת התהליך המחזיק אותו, אחרי שהתהליך הנ"ל סיים את משימתו.
4. **Circular wait**: קיימת קבוצה {P0, P1, ..., P0} של תהליכים ממתינים, כך ש-P0 ממתין למשאב שמוחזק ע"י P1, P1 ממתין למשאב שמוחזק ע"י P2, ..., Pn-1 ממתין למשאב שמוחזק ע"י Pn ו-Pn ממתין למשאב שמוחזק ע"י P0.

<u>שיטות פתרון ל-deadlock:</u>

1. מניעה: כל הזמן נבדוק במע' שהיא לא נכנסת למצב deadlock.
2. שבירת ה-**deadlock**: אם יש deadlock ננסה לתקן את המצב.
3. רוב מע' ההפעלה לא מטפלות ב-deadlock ואם קורה אז צריך להתחיל מחדש את המחשב וכו'.

Chapter 8:

חייבים להביא תוכנית מהדיסק לזיכרון הפנימי ולמקם אותה בתהליך כדי להריץ אותה.

Binding instructions and data to memory:

Address binding of instructions and data to memory addresses can be happen at 3 different stages:

1. **Compile time**: אם המיקום בזיכרון ידוע מראש ניתן ליצור קוד אבסולוטי. חייבים לקמפל מחדש את הקוד אם המיקום ההתחלתי משתנה.
   כלומר, הזמן שהכתובות נקבעות בזמן הקומפילציה (למשל: jump). בזמן הקומפילציה כבר קבעתי את הכתובות ולא ניתן יותר לשנות את מיקום התהליך בזיכרון הפנימי

ה-branch הוא יחסית לאיפה שאני נמצא ואילו jump הוא ספציפית לכתובת שאליה צריך לקפוץ (אבסולוטי). ב-jump הכתובת נכתבת בקוד עצמו ← מקשה מאוד, למשל, אם רוצים להזיז את התהליך למקום אחר. אנו בעצם אומרים שהתהליך חייב להיות במקום מסוים בזיכרון ויוצרים בכך <u>מגבלה</u> (כי אם אין מקום אז צריך להעיף תהליכים או להזיז אותם ואח"כ להחזיר אותם - אי אפשר יהיה להזיז תהליך כנ"ל למקום אחר שאולי התפנה אלא נצטרך לחכות עד שהמקום שבו הוא היה יתפנה).
לכן נרצה להשתמש בפקודות יחסיות.

2. **Load time**: אם מיקום הזיכרון אינו ידוע בעת זמן הקומפילציה חייבים לייצר relocatable code.
   הכתובות נקבעות בזמן העלאת התוכנית לזיכרון - עדיין לא מספיק, כי מה אם נרצה להזיז את התוכנית באמצע?
3. **Execution time**: אם בזמן ביצוע התהליך התהליך יכול לזוז ממקטע זיכרון אחד לאחר אזי ה-binding נדחה עד לזמן הריצה.
   ה-execution time נותן חופש גדול למע' ההפעלה ונשאף תמיד ל-execution time.

Dynamic loading:

רוטינות אינן מועלות לזיכרון לפני שקוראים להן. עד שלא קראתי להן הן יושבות בזיכרון. תוך כדי ביצוע התוכנית מתחילים להגדיל את קטע הטקסט של התהליך.
יש בכך ניצולת מרחב זיכרון טובה כי רוטינה שלא נשתמש בה אף פעם לא תועלה לזיכרון. שימושי כשצריך קטעי קוד גדולים לטיפול במצבים שקורים באופן לא תכוף.

<u>כתובות פיזיות ולוגיות:</u>

<u>כתובת לוגית:</u> נוצרת ע"י ה-CPU (נקראת גם כתובת וירטואלית).
<u>כתובת פיסית:</u> מיפוי הכתובת הלוגית: הכתובת שנראית ע"י הזיכרון הפנימי.
ה-CPU מכיר כתובת לוגית ולא מעניין אותו איפה היא נמצאת. כתובת לוגית לא חייבת להיות אותו דבר כמו הכתובת הפיסית.

ה-Memory Management Unit (MMU) הוא התקן חומרה שממפה את הכתובות הלוגיות לפיסיות.

- בשיטה זו הערך שבאוגר ה-relocation (אוגר שנועד לתהליכים של המשתמש בלבד ואומר לדלג על מע' ההפעלה) מתווסף לכל כתובת שנוצרת ע"י תהליך המשתמש בזמן שהיא נשלחת לזיכרון. ה-CPU יוצר כתובות מגודל 0 ומעלה. המיפוי יבוצע ע"י הוספת ה-relocation register ואז מגיעים לכתובת הרצויה.
- תוכנית המשתמש מתעסקת עם <u>כתובות לוגיות</u>. היא אף פעם אינה יודעת מה הכתובת הפיסית האמיתית.
- הכתובות הלוגיות בתהליך נשארות תמיד אותו דבר, בעוד הכתובות הפיסיות משתנות כל הזמן.

אם משנים את מיקום התהליך בתוך הזיכרון ← נצטרך מע' יותר משוכללת לטיפול בתרגום הכתובת הלוגית לפיזית.

Swapping:

ניתן להעביר תהליך באופן זמני מחוץ לזיכרון ל-backing store (=דיסק מהיר גדול מספיק כדי להכיל עותקים של כל תמונות הזיכרון של כל המשתמשים וחייב לספק גישה ישירה לתמונות אלו), ואח"כ להחזירו לזיכרון להמשך הביצוע.
<u>Roll out, roll in:</u> תהליך עם עדיפות נמוכה יותר מוצא החוצה כדי שתהליך עם עדיפות גבוהה יותר יוכל לעלות לזיכרון ולהתבצע.
חלק ניכר מה-swap time הוא transfer time. סך זמן ההעברה הוא יחסי ל<u>כמות</u> הזיכרון שמתחלפת.

Contiguous Allocation (הקצאה נושקת):

הזיכרון הפנימי מחולק ל-2 מחיצות:

- מע' ההפעלה שבד"כ מוחזקת בכתובות נמוכות בזיכרון יחד עם מערך הפסיקות.
- תהליכי המשתמש שמוחזקים בכתובות גבוהות בזיכרון.

Single partition allocation (DOS):

- משתמשים בשיטת אוגר ה-relocation כדי להגן על תהליכי משתמשים אחד מן השני, וכן משינוי הקוד וה-data של מע' ההפעלה.
- אוגר ה-relocation מכיל ערך של כתובת פיזית נמוכה ביותר. אוגר ה-limit מכיל טווח של כתובות לוגיות - כל כתובת לוגית חייבת להיות קטנה יותר מאוגר ה-limit.

יתרון: פשוט, ולכן DOS עבדה בשיטה זו כי היא לא הייתה מע' הפעלה מקבילית.
חסרון: אם יש 2 תהליכים למשתמש, אזי התהליך ה-1 יושב במחיצה והתהליך ה-2 בא ומוחץ אותו (כי יש רק מחיצה אחת).

Multiple partition allocation:

Hole - בלוק של זיכרון זמין. holes בגדלים שונים מפוזרים ברחבי הזיכרון.
כשמגיע תהליך הוא מקצה זיכרון מ"חור" גדול מספיק להכיל אותו.
אם אין חור מספיק גדול או אין חור בכלל ← מתחילים להוציא תהליכים החוצה.
מע' ההפעלה שומרת מידע על:

א. מחיצות שהוקצו.
ב. מחיצות (חורים) חופשיות.

אם גודל המחיצה הוא קבוע זה גורר בזבוז (internal fragmentation).

Dynamic Storage Allocation Problem: כיצד לספק בקשה מגודל n מתוך רשימה של חורים חופשיים.
3 שיטות:

- **First-fit**: להקצות את החור הראשון שגדול מספיק.
- **Best-fit**: להקצות את החור הקטן ביותר שהוא מספיק גדול. לשם כך צריך לחפש בכל הרשימה, אלא אם כן היא מסודרת לפי גודל. שיטה זו יוצרת את ה-leftover hole הקטן ביותר.
- **Worst-fit**: להקצות את החור הגדול ביותר. לשם כך צריך לחפש בכל הרשימה. שיטה זו יוצרת את ה-leftover hole הגדול ביותר. מדוע זה טוב? למשל: אם יש לנו מחיצה של 100K ורוצים להכניס תהליך של 2K אז נכניס ל-100K וההנחה היא ש-2K לא תופסים כ"כ הרבה מהמחיצה ולכן זה די זניח ונוכל להכניס לשם תהליך אחר.

ה-First fit ו-best-fit טובים מה-worst-fit מבחינת מהירות וניצולת מקום.
לא תמיד ה-best fit טוב יותר מה-first fit! בממוצע, first fit works best. הסיבה: best fit נוטה להותיר שטחים רבים של זיכרון שקטנים מדי לניצול.

Fragmentation:

External fragmentation (פיצול חיצוני): מחיצות בגודל משתנה. סה"כ יש הרבה מחיצות זמינות שמפוזרות ואם נחבר את כולן יחד נגיע לגודל הרצוי לתהליך, אבל אי-אפשר להכניס את התהליך לתוכם.
Internal fragmentation: מחיצות בגודל קבוע. תמיד מקצים גודל זהה של מחיצות ואם אין מספיק במחיצה אחת לתהליך אז מקצים כמה מחיצות. הפיצול הפנימי מתבטא בכך שיש מקומות שנחשבים כמוקצים למרות שהם לא מוקצים (למשל: תהליך של 0.5K במחיצות בגודל 8K ← 7.5K מבוזבז).

ניתן לצמצם פיצול חיצוני ע"י דחיסה:

- ערבול כל תוכן הזיכרון כך שנמקם את כל הזיכרון הפנוי יחד בבלוק אחד גדול.
- ניתן לבצע דחיסה רק אם ה-relocation היא דינמית ונעשית ב-execution time.
- בעיית I/O. פתרונות:

1. Latch job in memory while it is involved in I/O.
2. Do I/O only into OS buffers.

הסבר: פעולות הק/פ נעשות ע"י ה-kernel בלבד. עד שה-DMA מביא את הנתונים מהדיסק לוקח זמן. נניח שבזמן הזה החליטה מע' ההפעלה לבצע דחיסה, אז ה-DMA יכניס את הנתונים למקום לא נכון בזיכרון (למקום הישן).
פתרון:

1. נעילת ה-job בזיכרון כאשר הוא מעורב ב-I/O ← סיבוך הדחיסה.
2. לעשות את ה-I/O לתוך מע' ההפעלה. במחיצה של מע' ההפעלה אף אחד לא נוגע. ה-DMA כותב לתוך הזיכרון ב-kernel ואז מעתיק את הנתונים מה-kernel לתהליך (יוצר בזבוז).

:Paging

המטרה: להקצות מקום בזיכרון, אבל לאו-דווקא מקום רציף.
מקובל לתת את ההקצאות בחתיכות קבועות. מגדירים יחידת זיכרון - "דף". צריך לייצר טבלת דפים שתגיד לנו איפה כל דף יושב בזיכרון (וזה יכול להשתנות תוך כדי התוכנית - למשל אם צריך להוציא תהליך וכו').

- מחלקים את הזיכרון הפיסי לבלוקים בגודל קבוע שמכונים **frames** שגודלם הוא חזקה של 2.
- מחלקים את הזיכרון הלוגי לבלוקים בגודל זהה שמכונים **pages**.

כדי להריץ תוכנית בגודל n דפים, צריך למצוא n פריימים פנויים ולהעלות את התוכנית.
טבלת הדפים משמשת לתרגום כתובות לוגיות לפיסיות.
הדבר יוצר <u>פיצול פנימי</u> - כאשר מקצים דף שלם וצריך פחות מדף ← השאר מבוזבז.

<u>תרגום הכתובות (שקף 8.13, 8.14):</u>

הכתובת שנוצרת ע"י ה-CPU מחולקת ל:

- מס' דף (p) - משמש כאינדקס בטבלת הדפים שמכילה כתובת בסיס של כל דף בזיכרון הפיסי.
- היסט דף (d) - מצורף לכתובת הבסיס כדי להגדיר את כתובת הזיכרון הפיסי שנשלח ליחידת הזיכרון.

בד"כ הפנייה לטבלת הדפים נעשית אוטומטית ע"י התהליך ואין צורך במע' ההפעלה. צריך את מע' ההפעלה רק כאשר הנתונים אינם בדפים כי אז צריך להחליף את הדפים.

<u>מימוש טבלת הדפים:</u>

את טבלת הדפים שומרים בזיכרון הראשי.
ה-Page table base register (PTBR) מצביע לטבלת הדפים (האוגר נמצא בתוך ה-CPU).
ה-Page table length register (PTLR) מציין את גודל טבלת הדפים.
אם תהליך רוצה לדעת איפה הדף שלו הוא פונה ל-PTBR ומחשב את הכתובת.
סוף הטבלה = PTBR + PTLR.
בשיטה זו כל data/instruction access דורשת 2 גישות לזיכרון: אחת בשביל טבלת הדפים ואחת בשביל ה-data/instruction (בשביל למצוא את הנתון פיזית). כדי לפתור בעיה זו ניתן להשתמש בfast-lookup hardware cache המכונה associative registers או translation look-aside buffers (TLBs) - כעת טבלת הדפים תמיד ב-cache ← הפנייה מהירה.

<u>Associative Register (חומרה):</u>

אוגרים שמכילים את הדפים האחרונים שהשתמשנו בהם ובאילו מסגרות הם יושבים. הפנייה לאוגרים מאוד מהירה וההשוואות נעשות ב<u>מקביל</u>.
תרגום הכתובת עבור דף P1 נעשה כך:

- אם P1 נמצא ב-associative register אין פנייה לטבלת הדפים ופשוט מוציאים את מס' ה-frame.
- אחרת, הוצא את מס' ה-frame מטבלת הדפים בזיכרון.

כל פנייה לאוגר לוקחת ϵ יחידות זמן.
נניח שזמן ממוצע לפנייה לזיכרון הוא 1 מיקרו-שניות.
נגדיר <u>אחוזי פגיעה</u> כאחוז הפעמים שמס' דף נמצא ב-associative registers ונסמן אותו כ-α.
לכן ה-Effective Access Time (EAT): $EAT = (1+\varepsilon)\alpha + (2+\varepsilon)(1-\alpha) = 2 + \varepsilon - \alpha$
<u>הסבר:</u> במקרים שהייתה פגיעה יש 1 - הפנייה עצמה לזיכרון והפנייה לאוגר - ϵ.
במשלים $(1-\alpha)$ יש פנייה לאוגר (ϵ) ויש תשובה שלילית ואז יש 1 - פנייה לזיכרון לטבלת הדפים ועוד 1 - להוצאת הנתון עצמו.

<u>Two Level Page Table Scheme: (שקף 8.18, 8.19)</u>

כיוון שיש הרבה מסגרות ← טבלת הדפים תתפרש על המון שטחים.
<u>פתרון:</u> 2 רמות:

1. טבלה 1 שאומרת לנו שדף מסוים נמצא בין frame a ל-frame b.
2. טבלה 2: הפנייה מדויקת ל-frame המסוים בו נמצא הדף.

כלומר, יש 2 רמות עד שמגיעים לזיכרון הפיזי עצמו.
פתרון זה יוצר טבלת דפים קטנה יותר.
בשיטה הישנה כשנוצר חור באמצע - החור היה נשאר שם. בשיטה זו הטבלה מחולקת לתת-דפים ואם נוצר חור אזי ניתן להכניס לשם נתונים אחרים ← הטבלה תהיה קטנה יותר.

כיון שכל רמה מאוחסנת כטבלה נפרדת בזיכרון מעבר מכתובת לוגית לפיסית עשויה ליצור יותר גישות לזיכרון.
למרות שהזמן הדרוש לגישה אחת לזיכרון הוא פי 5, ה-caching מאפשר לביצועים להישאר סבירים במידה שה-cache hit יהיה גבוה.

Shared pages: (8.22)

קוד משותף:
עותק אחד של read-only code משותף בין התהליכים (למשל: text editors). הקוד המשותף חייב להופיע באותו מיקום במרחב הכתובות הלוגיות עבור כל התהליכים.
כל תהליך שומר עותק נפרד של ה-code and data. הדפים עבור ה-code and data הפרטיים לכל תהליך יכולים להופיע בכל מקום במרחב הכתובות הלוגיות.

Segmentation:

שיטת ניהול זיכרון שתומכת במבט המשתמש על הזיכרון.
תוכנית היא אוסף של מקטעים. מקטע מכיל קטע לוגי מהתהליך (למשל: תוכנית ראשית, פרוצדורה, פונקציה, משתנים, מחסנית, מערכים). כלומר, במקום לחלק את התוכנית ל-pages רוצים להיות גמישים ולקבוע גודל אחר לגודל הדף ← segmentation. את התוכנית מחלקים באופן לוגי למקטעים בגדלים שונים (קיטוע).
עדיין יכול להיווצר פיצול חיצוני כי עדיין יש חורים כיוון שאם היה מקטע בגודל 100 למשל, ואז בא תהליך בגודל 98 - מכניסים אותו למקטע בגודל 100, אולם יש בזבוז של 2 בתים.
יתרון: הקיטוע יותר גמיש וניתן לקבל גדלים שונים (כלומר, לא חייבים גדלים קבועים כמו ב-pages).
חסרון: הניהול שלו יותר קשה.

ניהול מקטוע:
כתובת לוגית מורכבת ממס' סגמנט ומהיסט.
בניגוד לטבלת הדפים, טבלת המקטעים יותר מורכבת כיוון שצריך להגיד את הכתובת המסוימת ולא ניתן לעשות הכפלות (כי הגדלים משתנים). לכל שורה בטבלה יש:
base: מכיל את תחילת הכתובת הפיסית היכן שהמקטעים יושבים בזיכרון.
limit: מכיל את גודל המקטע (ואילו בטבלת הדפים גודל כל דף קבוע).
ה-Segment-table base register (STBR) מצביע למיקום טבלת המקטעים בזיכרון.
ה-Segment table length register (STLR) מציין כמה מקטעים מנוצלים ע"י התוכנית.
מס' מקטע הוא חוקי רק אם הוא קטן מ-STLR.
סוף הטבלה = STBR + STLR.

שיתוף מקטעים: (8.26)
את הקוד המשותף נשים במקטע אחד ונמפה אותו לאותו מקום בזיכרון.

Chapter 9:

זיכרון וירטואלי:

Virtual memory: separation of user logical memory from physical memory.
רק חלק מהתוכנית צריכה להיות בזיכרון בשביל הביצוע. לכן מרחב הכתובות הלוגיות יכול להיות גדול יותר ממרחב הכתובות הפיסיות.

Need to allow pages to be swapped in and out.

ניתן ליישם זיכרון וירטואלי ע"י 2 דרכים:
demand paging או demand segmentation.

I. Demand Paging:

מביאים דף לזיכרון רק כאשר הוא נחוץ. יתרונות:

- צריך פחות **I/O**: אם היינו מביאים את כל התהליך אזי ה-paging היה יותר ארוך - כי צריך לחפש בכל מקום בדיסק, ואם הוא מלא - אזי להוציא/להכניס שוב ושוב.
- צריך פחות מקום בזיכרון.
- תגובה מהירה יותר: פחות paging, לכן רוב הסיכויים שדף מסוים יהיה כבר בזיכרון ולא נצטרך לחפש הרבה.
- יותר משתמשים: כשלא מביאים את כל הדפים בתהליך ← ניתן להכניס עוד תהליכים.

כשצריכים דף פונים אליו:
אם הפנייה לדף נגרמה עקב שגיאות (למשל: המחשב מתרגם כתובת פיסית 50 כאשר יש רק 40 ← invalid reference) ← abort.
אם הדף אינו בזיכרון ← מביאים אותו לזיכרון.

בטבלת הדפים מוסיפים **valid-invalid bit**, כאשר 1 ← הדף בזיכרון, 0 ← הדף לא בזיכרון. בהתחלה כל הביטים הם 0.
הביט אומר האם ה-frame בזיכרון באמת מכיל דף מהדיסק. אם frame משתחרר לא מכבים את הביט שהוא 1 אלא שמים במקומו ישר את הדף החדש. <u>ה-0 נמצאים רק בהתחלה</u>.
כל page יכול להיות בכל frame בזיכרון. אם הדף לא נמצא (כתוב 0 או שיש דף אחר שיושב באותו frame) ← page fault. ה-page fault הוא פסיקה ← יש קריאה למע' ההפעלה.
page fault תמיד נוצר בעת קריאה ראשונה לדף. מע' ההפעלה קודם בודקת אם יש invalid reference ואם כן אז abort, אחרת היא יוצרת page-fault. בעקבות פסיקה זו מחפשים frame ריק. אם יש אז שמים את הדף בתוך ה-frame שמצאנו ומדליקים את ה-valid bit. כעת ה-frame מכיל את הכתובת האמיתית.
אם אין frame פנוי אז מוצאים איזשהו דף בזיכרון שלא ממש משתמשים בו (למשל: תהליך שמת והמסגרות שלו נשמרו, או דף שכן בשימוש ופחות חשוב וכנראה לא נפנה אליו בקרוב) ומחליפים ביניהם. כיוון שכל page fault גורם לפנייה למע' ההפעלה שמביאה דפים למע' הזיכרון ← <u>עיכוב</u>, לכן מחפשים אלגוריתם שיגרום להכי פחות page faults.

נסמן ב-p את ה-page fault rate: $0 \le p \le 1$
אם p = 0 אז אין בכלל page faults (זיכרון מספיק גדול), לכן נשאף תמיד לכך ש-p ישאף ל-0.
אם p = 1 אזי כל פנייה לדף היא fault (זיכרון עם מסגרת אחת בלבד).
נגדיר את ה-Effective Access Time for One Page (EATOP):

EATOP = (1-p) * memory access {page fault לא היה←במקרים שהייתה פנייה לזיכרון בלבד}
+p(page fault overhead + [swap page out if needed] + swap page in + memory access)

<u>הסבר שורה 2:</u> במקרים שכן היה page fault:
page fault overhead: לפי אלגוריתם LRU: חישובים למציאת הקורבן שנחליף.
[swap page out if needed]: אם הדף הובא רק לקריאה - אין צורך לכתוב אותו חזרה לדיסק כשמעיפים אותו. רק אם היה שינוי בדף נכתוב אותו חזרה לדיסק.
swap page in: זמן הבאת הדף לזיכרון.
memory access: פנייה לזיכרון והבאת הנתונים (הזמן המינימלי ביותר מבין כל שאר הגורמים הנ"ל).

משתמשים ב-**modify (dirty) bit** כדי לצמצם את ה-overhead בעת החלפות דפים ← רק דפים ששונו ייכתבו חזרה לדיסק.
Page replacement completes separation between logical memory and physical memory - large virtual memory can be provided on a smaller physical memory.

אלגוריתמים עבור page replacement:

רוצים page-fault מינימלי.

1. First In First Out (FIFO) (שקף 9.9)

שימו לב:

FIFO Replacement - **Belady's Anomaly: more frames ⇏ less page faults**

2. Optimal Algorithm (פתרון בלתי מעשי) (שקף 9.10):

החלפת הדפים היא לפי הדפים שלא ייעשה בהם שימוש לתקופת הזמן הכי ארוכה. אולם לא ניתן לדעת זאת מראש. שיטה זאת אופטימלית והיא משמשת כדי לבדוק את טיב האלגוריתם שכן משתמשים בו.

3. Least Recently Used (שקף 9.11):

בדומה ל-FIFO, אבל, בנוסף, לכל דף יש תווית זמן וכל פנייה לדף ← מעתיקים את זמן השעון לתוך תווית הזמן שלו. כאשר צריכים להוציא דף מסתכלים על תוויות הזמן ולדף שיש תווית זמן הכי נמוכה ← אותו מוציאים.

4. Counting Algorithms:

מחזיקים מונה למס' הפעמים שפנינו לכל דף.

2 אלגוריתמים:

א. LFU (Least Frequently Used): מחליפים את הדף שהמונה שלו הכי קטן.

ב. MFU (Most Frequently Used): מחליפים את הדף שהמונה שלו הכי גבוה: מסתמכים על ההנחה שדף עם המונה הכי קטן רק הגיע לזיכרון עכשיו ועוד לא השתמשו בו.

איך מקצים את הזיכרון?

1. Fixed Allocation:

הקצאה שווה: למשל: אם יש 100 frames ו-5 תהליכים אזי כל תהליך יקבל 20 דפים.

הקצאה יחסית (שקף 9.13): ההקצאה נעשית בהתאם לגודל התהליך. כלומר:

אם Si הוא גודל תהליך Pi, S הוא סכום כל הגדלים ו-m הוא מספר ה-frames אזי ההקצאה לכל תהליך:

$$a_i = \frac{s_i}{S} \bullet m$$

2. Priority Allocation:

לכל תהליך יש עדיפות. כשיש page fault מחפשים frame של תהליך עם עדיפות נמוכה יותר ואותו מעיפים. אם אין אף frame כזה אז נבחר להעיף אחד מה-frames של התהליך עצמו או של תהליך אחר עם אותה עדיפות. אם גם אין אף frame כזה המע' תעדיף לא להכניס אותו, אולם הוא ייכנס ונבחר להעיף frame של תהליך עם עדיפות גבוהה יותר (סביר להניח שעקב ההכנסה ייגרם page fault).

3. Global & Local Allocation:

החלפה גלובלית: התהליך בוחר להחליף frame מקבוצת כל ה-frames שיש. כך תהליך יכול לקחת frame מתהליך אחר.

החלפה מקומית: כל תהליך בוחר frame רק מקבוצת ה-frames שלו.

ה-global & local allocation לא מתחשבים בכל הפרמטרים. הם משתמשים בשיטת LRU ← פחות overhead.

ה-global מחפש LRU בכל ה-frames ואילו ה-local מחפש קודם LRU בדפים של התהליך עצמו, ורק אם אין דפים מיותרים שהתהליך לא צריך מהתהליך עצמו הוא עובר לחפש בשאר הדפים.

עקרון המקומיות:

כאשר מביאים value, יש סבירות גבוהה שנצטרך גם את "השכנים" שלו. כלומר, כשמביאים page שלם ← מביאים נתונים שכנראה נצטרך בקרוב, ולכן הבאת כל ה-page הגיונית:

Why does paging work? - Process migrates from one locality to another.
- Localities may overlap.

Thrashing (דשדוש):

דשדוש = תהליך עסוק בהחלפת דפים מחוץ/לתוך הזיכרון. הדשדוש קורה בגלל שסכום גודל המקומיות גדול מסך המקום בזיכרון.

אם יש הרבה תהליכים וכולם דורשים המון זיכרון ויש להם שכונות (locality) שאיתן הם עובדים ואין מספיק זיכרון אז מתחילים כל הזמן להוציא ולהכניס ← כמעט כל פנייה לזיכרון היא page-fault.
ככל שמוסיפים עוד תהליכים ← ניצולת ה-CPU עולה (ה-CPU מתחיל לעבוד יותר ויותר קשה) ובשלב מסוים תהיה נפילה, כיוון שנצטרך להביא כל הזמן דפים ולהחזיר דפים וה-CPU במקום לעבוד רק מחכה להוצאת הדפים והחזרתם (דשדוש).

<u>שיקולים אחרים בבחירת הדפים:</u>

1. fragmentation.
2. גודל הטבלה: נרצה שגודל טבלת הדפים תהיה כמה שיותר קטנה (כי היא תופסת מקום מבוזבז בזיכרון) ← הגדלת גודל דף.
3. I/O overhead: ה-DMA מביא בלוקים שלמים מהדיסק. לא נגדיר גודל דף גדול מדי כי אחרת כל הבאה של דף תיקח הרבה זמן (בד"כ נגדיר כ-1 בלוק של DMA).
4. locality.
5. שקף 9.19.

<u>II. Demand Segmentation:</u>

אותו רעיון כמו demand paging רק שהמקטע הוא בגודל משתנה.
ה-segment descriptor מכיל valid bit שמציין האם המקטע נמצא בזיכרון. אם המקטע בזיכרון אז ממשיכים. אם הוא לא בזיכרון ← segment fault.

Chapter 13:

מבנה הדיסק:

התקני דיסק ממופים כמערכים גדולים (חד-מימדיים) של בלוקים לוגיים, כאשר בלוק לוגי הוא היחידה הקטנה ביותר של העברה. מערך זה ממופה למקטעים של הדיסק באופן סידרתי. המיפוי מתחיל מהמסילה הראשונה עד המסילה האחרונה בצילינדר הראשון ואז לשאר הצילינדרים.

מע' ההפעלה אחראית לשימוש בחומרה באופן יעיל ← זמן גישה מהיר לדיסק. 2 מרכיבים עיקריים לזמן הגישה: seek time (זמן החיפוש: הזמן שלוקח לדיסק להגיע לראש הצילינדר שמכיל את המקטע המבוקש) ו-rotational latency (זמן חיפוש סיבובי).
המטרה היא שזמן החיפוש יהיה מינימלי. ה-seek time שווה בערך ל-seek distance.

אלגוריתמים לתזמון הדיסק:

1. **First Come First Served (FCFS)** (שקף 13.4)
2. **Shortest Seek Time First (SSTF)** (שקף 13.6):

לאן הכי קרוב ללכת: בוחרים את הבקשה שזמן ה-seek time מהמיקום הנוכחי אליה הוא המינימלי.
ה-SSTF דומה ל-SJF. בשיטה זו ה-seek time יותר קטן באופן ממוצע.
יכול להיווצר **starvation**: אם כל הזמן מגיעות בקשות מאזור צילינדר מסוים ואילו יש בקשה שנמצאת באזור מרוחק ← כל הזמן נישאר באזור ולא נגיע לבקשה היחידה. בד"כ, הנהנים העיקריים משיטה זו הם הצילינדרים האמצעיים.
כיוון שבד"כ הדיסק אינו עמוס ה-SSTF הוא די טוב.

3. **SCAN = Elevator Algorithm** (שקף 13.8):

זרוע הדיסק מתחילה מקצה אחד של הדיסק וזזה לכיוון הקצה השני תוך כדי שירות בקשות. כאשר מגיעים לצד השני הולכים בחזרה וכן הלאה.
בשיטה זו יש עדיין עדיפות לצילינדרים האמצעיים כי עוברים בהם פעמיים בכל כיוון. למרות שאין הרעבה עובר זמן די קבוע עד שמגיעים לכל מקום.

4. **C-SCAN** (שקף 13.10):

זרוע הדיסק נעה מצד לצד. השוני בין שיטה זו ל-SCAN הוא שכשהזרוע מגיעה לצד השני היא חוזרת מיד לתחילת הדיסק בלי לשרת אף בקשה בחזרה ← אין תהליכים שמקופחים, כולם מקבלים אותו שירות.
Treats the cylinder as a circular list that wraps around from the last cylinder to the first one.
ל-SCAN ול-C-SCAN יש ביצועים טובים יותר במע' עם עומס כבד על הדיסק.

5. **C-LOOK** (שקף 13.12):

גירסה של C-SCAN. הזרוע הולכת רק עד המיקום הכי רחוק של הבקשות ואז חוזרת ישר להתחלה.
באותו אופן יש את LOOK שהיא גירסה של SCAN שבזמן החזרה כן משרתת בקשות נוספות.
יכול להיות שמישהו בצילינדר האחרון יהיה מקופח כי הוא יגיע לשם פחות. מה גם שבד"כ הדיסקים אינם מלאים עד הסוף והצילינדרים האחרונים אינם מלאים.

הערה: הביצועים תלויים במס' הבקשות וסוגיהם. בקשות לשירות דיסק עשויות להיות מושפעות לפי שיטת הקצאת הקבצים (שמשפיעה גם על ניהול הדיסק).
אם סוג הבקשות הוא לאותו צילינדר ← אין בעיה ← כשמחלקים דיסק מחלקים אותו באופן הגיוני.