

יסודות מערכות הפעלה מבוא למערכת ההפעלה Linux

מהדורה 2.2, June 2017

ד"ר סמי זעפרני



Image courtesy of Computer History Museum, www.computerhistory.org

תוכן עניינים

7	1 מבט כללי	
7	1.1 נושאי הלימוד	1.1
8	1.2 ספרי לימוד	1.2
8	1.3 מושגי יסוד	1.3
14	1.4 מערכות פתוחות ומערכות סגורות	1.4
18	2 סקירה היסטורית	
19	2.1 הדור הראשון	2.1
21	2.2 הדור השני 1955-1965	2.2
24	2.3 הדור השלישי 1965-1980	2.3
26	2.4 מערכות שיתוף זמן (Time-Sharing Systems)	2.4
29	2.5 הדור הרביעי 1980-1995	2.5
33	3 המבנה של מערכת הפעלה	
34	3.1 ניהול תהליכים	3.1
36	3.2 ניהול זיכרון	3.2
37	3.3 ניהול התקני איחסון משניים	3.3
38	3.4 ניהול מערכת קלט/פלט	3.4
39	3.5 ניהול מערכת קבצים	3.5
41	3.6 הגנה	3.6
41	3.7 תיקשורת בין מחשבים שונים	3.7

43	מעבד פקודות - Shell	3.8
47	מבנה של מערכת מחשב	4
47	פעולה שגרתית של מחשב	4.1
51	קלט ופלט	4.2
53	DMA - Direct Memory Access	4.3
55	מבנה האיחסון	4.4
59	הגנה ברמת החומרה	4.5
61	ניהול הגנת זיכרון	4.6
64	קריאות שרות (system calls)	4.7
66	מערכת ההפעלה Unix	5
66	היסטוריה	5.1
74	קריאות שירות (System Calls)	5.2
77	Unix - מושגי יסוד	6
77	פקודות שימושיות וסימנים מיוחדים	6.1
79	קבלת עזרה	6.2
81	תהליכים	6.3
87	מערכת הקבצים	6.4
98	זיכרון וירטואלי	7
99	משימות עקריות בניהול זיכרון	7.1
101	מרחב כתובות וירטואלי	7.2
104	דיפדוף (Paging)	7.3
112	מדיניות פינוי דפים	7.4
115	הערות מסכמות	7.5

118	מעבד הפקודות Bash	8
121	תהליך העיבוד	8.1
124	פקודות פשוטות	8.2
128	קישור פקודות בעזרת אופרטורים	8.3
130	הפניית קלט ופלט	8.4
133	הפקודה test	8.5
137	פקודות מורכבות	8.6
141	שלושה אופני ביצוע	8.7
145	שלושה סוגי פרמטרים	8.8
150	שלושה אופני ציטוט	8.9
152	שלושה אופני המרה	8.10
158	קיצורים (aliases)	8.11
158	בקרת משימות (Job Control)	8.12
160	History and fc commands	8.13
161	פקודות פנימיות שכיחות	8.14
167	משתנים סביבתיים בשימוש המעבד	8.15
171	קובצי איתחול וסיום	8.16
184	ביטויים רגולריים	9
184	(Regular Expressions)	9.1
188	תתי-ביטויים (Subexpressions)	9.2
190	ביטויים רגולריים מורחבים	9.3
192	חיפוש והחלפת ביטויים רגולריים	9.4
193	ביטויים רגולריים ב-Awk	9.5
198	פיתרון תרגילים	10

הקדמה

חוברת הקורס מתבססת באופן הדוק על הספר Operating System Concepts שחובר על יד Silberschatz/Galvin ועל סדרת שקפים של קורסי מערכות הפעלה ומבוא ל-UNIX אשר ניתנו במכללת נתניה בשנים 1997-2000. אנו מודים לאבי זילברשץ על הסכמתו לאפשר לנו להשתמש בתרשימים וטבלאות מתוך הספר למטרות הקורס. חלק ניכר של התרשימים מקורו במערכת השקפים המלווה את הספר של Galvin ו-Silberschatz, וחלק גדול של החומר מתבסס באופן הדוק על הפרקים הראשונים של הספר. כמו־כן אנו מודים מאוד למוזיאון ההיסטוריה של המחשבים ארה"ב <http://www.computerhistory.org> על הצילומים ההיסטוריים הנפלאים, שבאדיבותו הרבה נתן לנו את הרשות להשתמש בהם בחוברת הזו. ניתנת בזאת הרשאה לכל המעוניין להוריד, ולהפיץ את החוברת עבור מטרות הוראה בלבד, ובתנאי שלא ייתבצע שום שינוי ולא ייגבה תשלום כלשהו עבור החוברת.

עקב הזמן הרב שעבר מאז שהוכנו השקפים המקוריים, חלק גדול מהנושאים זקוקים לריענון, עידכון, והשלמה. כמו־כן, עדיין חסרים מספר נושאים בתוכנית הלימודים שלא נכללו עדיין. במהדורות הבאות של החוברת יעשה מאמץ לעדכן את החוברת ולהתאים אותה להתפתחויות האחרונות. למרות זאת, מרבית החומר עדיין בתוקף והחוברת רובה ככולה עדיין מתאימה לקורסי יסוד במערכות הפעלה ובפרט מערכת ההפעלה Unix.

סמי זעפרני, March 2014, June 2017

פורמט

החוברת הוכנה עבור טבלטים וקוראים אלקטרוניים לסוגיהם הרבים, ואין שום תוכנית או כוונה להוצאתה בפורמט של ספר מודפס על גבי נייר רגיל. פשוט יהיה חבל מאוד על העצים שיידרשו לשם כך. מסיבה זו גודל הגופן שנעשה בו שימוש הוא גדול במיוחד, וזאת בכדי להקל על הקוראים, בפרט במכשירים האלקטרוניים הקטנים שבהם קריאת חומר כתוב עשויה להיות קשה (במיוחד למבוגרים שבינינו). סיבה נוספת לפורמט המוגדל היא בכדי לאפשר למורים להשתמש בדפי החוברת כשקפים בקורס. החוברת ברובה התפתחה מתוך שקפים שניתנו בקורס זה בעבר, והמעבר לפורמט מורחב ומפורט של ספר רגיל לא נראה כרעיון טוב ומתאים לזמנים החדשים. אנו מקווים כי למרות התימצות המתחייב מסגנון עריכה זה, החוברת תהיה מועילה עבור סטודנטים בקורסי מערכות הפעלה למיניהם הרבים.

פרק 1

מבט כללי

1.1 נושאי הלימוד

- מושגי יסוד בתחום מערכות הפעלה: מוניטור, משאבי מערכת מחשב כגון זיכרון, יחידת עיבוד, דיסקים, וכו'
- עקרונות מערכות פתוחות והשימוש בהן
- ריבוי תהליכים וסביבת עבודה מרובת משתמשים ותהליכים
- ויסות מעבד וחלוקת משאבי המערכת בין המשתמשים
- בקרת קבצים, ניהולם, מבנה מערכת הקבצים, ומבני הנתונים המתחזקים אותם
- אבטחת מידע, הרשאות גישה, נעילות, הגנות בתוכנה
- מערכת ההפעלה Unix כדוגמא למערכת פתוחה

1.2 ספרי לימוד

1. Operating System Concepts, A. Silberschatz and P. Galvin
2. Modern Operating Systems, Andrew Tanenbaum
3. The Unix Programming Environment, B.W. Kernighan and R. Pike
4. Unix for the impatient, P.W. Abrahams and B.R. Larson
5. The design of the Unix operating system, Maurice Bach
6. Advanced Programming in the Unix Environment, W.R. Stevens

1.3 מושגי יסוד

- **מערכת הפעלה** היא תוכנית מחשב המתווכת בין המשתמש לבין החומרה והציוד ההיקפי של המחשב.
- מטרת מערכת ההפעלה היא לספק למשתמש סביבה שבה הוא יוכל להפעיל תוכניות מחשב רגילות.
- הדרישה העקרית ממערכת ההפעלה היא להפוך את מערכת המחשב (חומרת המחשב והציוד ההיקפי הנלווה) **נוחה** ככל האפשר לשימוש. דרישה משנית היא לאפשר את השימוש במערכת המחשב בצורה יעילה ככל האפשר.

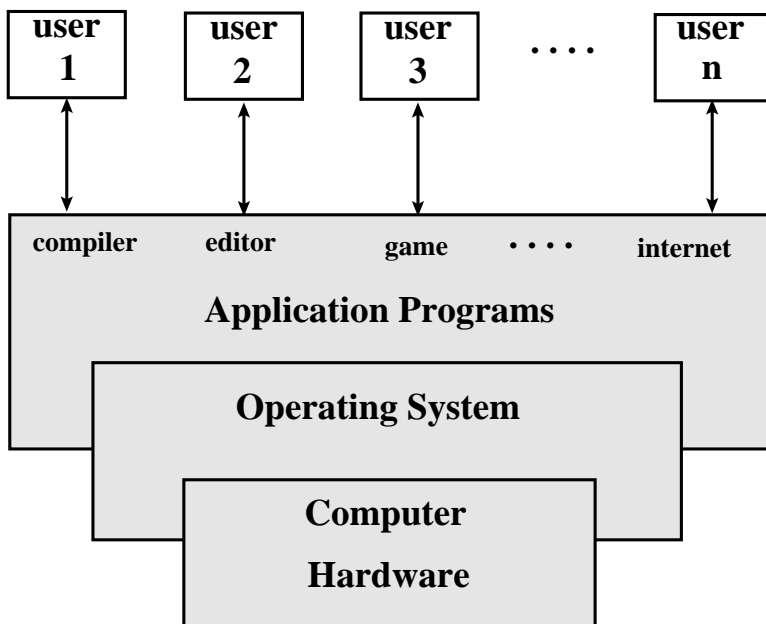
- בכדי להבין מהי מערכת הפעלה, הכרחי להבין כיצד מערכות הפעלה התפתחו מבחינה היסטורית. מערכות הפעלה פותחו (ויפותחו) בכדי לתת פיתרונות לבעיות רבות שהתעוררו במהלך השימוש השוטף במחשבים מימיהם הראשונים ועד היום.
- המשתמש המודרני אינו מודע לרוב הבעיות האלה בזכות העובדה שעומדים לרשותו מערכות הפעלה מצוינות הפותרות אותן ומסתירות מפניו את המורכבות והסיבוך של המערכת בה הוא משתמש.
- מערכת מחשב מורכבת בצורה גסה מארבעה חלקים:

א. חומרה וציוד היקפי

ב. מערכת הפעלה

ג. תוכניות יישומים

ד. משתמשים



איור 1.1: מערכת מחשב

● חומרה

רכיבי חומרה בסיסיים הם

א. Central Processing Unit – CPU

ב. זיכרון

ג. התקני קלט ופלט כגון צג, מקלדת, עכבר, מדפסת, סורק תמונות, רשם קול, כונני דיסקים וכונני CDROM, ועוד.

● תוכניות יישומים

יישומים שכיחים הם:

א. קומפיילרים עבור שפות תיכנות. סביבות פיתוח כגון Mi-

Java Netbeans, Eclipse, crosoft Visual Studio.

ב. עורכי ומעבדי תמלילים כגון word, notepad, edit, וכו'

ג. תוכנות גרפיות כגון Adobe Photoshop, Microsoft Visio

ד. גיליונות אלקטרוניים כגון Excel

ה. דפדפני אינטרנט כגון Chrome, Firefox, או MS Explorer.

ו. תוכנות תיקשורת להעברת או טעינת קבצים בין מחשבים

שונים, כניסה למחשבים מרוחקים, הפעלת תוכניות מרחוק,

מישלוח וקבלת דואר אלקטרוני או והודעות.

ז. מערכות בסיסי נתונים כגון MS SQL, MS-Access, Oracle.

ח. משחקים.

● מערכת הפעלה

כאמור לעיל, לא ניתן להפעיל תוכניות יישומים שונות ולנצל את החומרה והציוד ההיקפי של המחשב ללא מערכת הפעלה מתאימה שתאפשר זאת. קיים מספר רב של מערכות הפעלה. להלן כמה דוגמאות מוכרות:

- א. גירסאות שונות של Dos.
- ב. Windows 95, NT, 2000, XP, Vista של חברת Microsoft
- ג. Windows 7, 8, 10, של חברת Microsoft
- ד. גירסאות שונות של: Linux, Unix
- ה. Android (Google)
- ו. ChromeOS (Google)
- ז. OS/2 של חברת IBM
- ח. VMS של חברת IBM
- ט. סדרת MacOS של חברת Apple

● משתמשים

למחשב יכולים להתחבר מספר גדול של משתמשים.

- א. המשתמשים הם בדרך כלל בני אדם באמצעות צגים שונים המחברים לאותו המחשב.

ב. מחשבים אחרים יכולים להתחבר אל מחשב מסוים לשם קבלת שרותים שונים.

ג. מכונות ייצור, מכשירים רפואיים, חיישנים (sensors), וכו'.

- מערכת ההפעלה צריכה לספק את האמצעים לשם הפעלה נאותה וניצול יעיל של המשאבים השונים שבמערכת המחשב. במובן זה, מערכת המחשב דומה לממשלה שאמורה לספק לאזרחים (המשתמשים) את השרותים הדרושים בצורה מרוכזת, יעילה, וצודקת.

- ניתן לומר כי מערכת ההפעלה היא יישום היוצר סביבה נוחה שבה יישומים אחרים יוכלו לפעול בצורה תקינה ויעילה.

- אפשר לתאר את מערכת ההפעלה כמנגנון לחלוקה צודקת וחכמה של משאבי המחשב העומדים לרשות המשתמשים. בכל מערכת מחשב נתונה, קיימים גבולות ברורים למשאבים בה: כמות הזיכרון, שטח האיחסון בכוננים, מהירות המעבד, מהירות התקני הקלט והפלט, וכדומה. מערכת ההפעלה היא המנהל של המשאבים האלה והיא צריכה לדאוג לחלוקה הוגנת וחכמה שלהם בין המשתמשים השונים בהתאם למעמד, דחיפות, ולמידה הדרושה לשם ביצוע משימותיהם.

- נציין כי דרישת "היעילות" ודרישת "ההוגנות" עשויות להתנגש אחת בשניה, מה שמקשה יותר על תיכנון נכון של מערכות הפעלה. לשם מילוי דרישת ההוגנות, על מערכת ההפעלה

לבזבז זמן ומשאבים יקרים על איסוף ושמירת נתונים לשם חישוב זמנים, תורים, וכמויות ניצול של המשאבים השונים בין המשתמשים השונים.

- תפקיד חשוב נוסף שיש למערכת ההפעלה הוא **בקרה ופיקוח** (באנלוגיה לממשלה זהו תפקיד המשטרה). על מערכת ההפעלה לפקח על תוכניות המשתמשים ולדאוג לכך שלא ייתבצעו פעולות מזיקות על ידיהן. במידה ובכל זאת התבצעו פעולות כאלה, על מערכת ההפעלה לזהות אותן ולמסור דו"ח מתאים לאחראי על המחשב. במיקרים רבים מערכת ההפעלה צריכה גם לספק כלים מתאימים לשם תיקון פגמים או תיחזוק של מערכת המחשב.
- למרות כל האמור לעיל, לא ניתן להגדיר בצורה ברורה וחד-משמעית מהי מערכת הפעלה! מערכות הפעלה קיימות בכדי לגשר על הפער שבין הגורם האנושי (המשתמש השכיח) לבין המורכבות ההולכת וגדלה של חומרת המחשב. מאחר שגם הגורם האנושי וגם החומרה נמצאות בתנועה מתמדת, הרי שגם תפקידי מערכת ההפעלה משתנים בהתאם.
- לכן הדרך היחידה לתאר מערכת הפעלה היא על ידי איפיונים כלליים ומופשטים, אשר גם עליהם אין תמיד הסכמה כללית ומוחלטת.
- הגדרה שכיחה אחת של מערכת הפעלה, למשל, היא: מערכת הפעלה היא התוכנית היחידה שרצה כל הזמן על המחשב

מתחילת פעולתו ועד לכיבויו (תוכנית זו נקראת בדרך כלל "גרעין"). שאר התוכניות שעולות ויורדות מזמן לזמן נקראות יישומים. אך גם הגדרה זו אינה לגמרי מדויקת, משום שקיימות היום מערכות הפעלה המורכבות ממספר חלקים שלא כולם נמצאים תמיד בזיכרון (הם עולים ויורדים במידת הצורך).

1.4 מערכות פתוחות ומערכות סגורות

- נהוג להבחין בין מערכות הפעלה פתוחות ומערכות הפעלה סגורות. אך גם כאן ההבחנה אינה חדה לגמרי. מערכת פתוחה היא מערכת המאפשרת למשתמש מידה רבה של חופש לשנות את תצורת המערכת ולעצב את סביבת העבודה שלו בצורה הרצויה לו. כלומר, למשתמש יש את האפשרות להתערב באופן הפעולה התקני של המערכת על ידי שינוי, יצירה, או הסרה של המאפיינים הסטנדרטיים של המערכת.

- מערכת הפעלה סגורה היא מערכת החוסמת בפני המשתמש כל דרך לשנות את אופן פעולתה. המערכת למעשה מסתירה את עצמה מפני המשתמש עד כדי כך שהמשתמש אפילו אינו מודע לקיומה. בכך מושגות המטרות הבאות:

א. **שקיפות:** מערכת ההפעלה שקופה למשתמש. המשתמש אינו מודע לקיומה של מערכת ההפעלה ולכן אינו זקוק להבין את המבנה וכל העקרונות והפרטים המורכבים

הקשורים לאופן פעולתה. לכן המשתמש יכול להתרכז רק בתוכניות היישומים אותם הוא מעוניין להפעיל. בכך נחסכים למשתמש המון זמן ואנרגיה הדרושים להכרת המערכת.

ב. **בטיחות:** המערכת מוגנת באופן מקסימלי מפני האפשרות של גרימת נזק בשגגה או בזדון מצד המשתמשים או מצד עצמה (במידה והיא מתוכננת לכך בצורה נכונה).

ג. **קביעות:** תצורת המערכת היא קבועה ולכן המשתמש לא יופתע על ידי התנהגות חריגה של המערכת בכל פעם שהוא נכנס אליה. (תצורת המערכת תהיה ניתנת לשינוי רק על ידי הספק או האדם המוסמך לכך).

ד. **יציבות ואמינות:** חוסר האפשרות של המשתמשים להתערב באופן פעולתה של המערכת מגדיל בצורה משמעותית את סיכוייה להשאר יציבה ואמינה לאורך זמן.

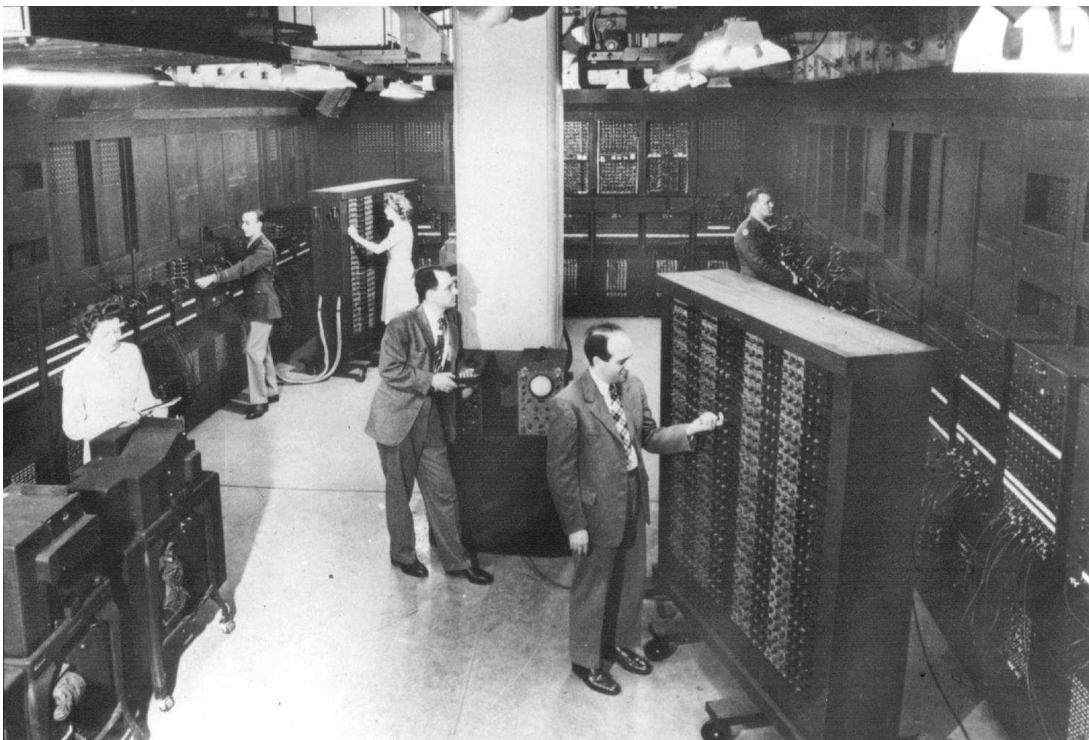
- דוגמא טובה של מערכת סגורה היא מערכת ההפעלה של מחשבי Macintosh. מחשבים אלה חסינים בפני וירוסים!
- היתרונות הנ"ל של מערכת סגורה הופכים אותה לאידיאלית עבור המשתמש הפשוט, המעוניין רק בהפעלה תקינה ויציבה של מספר יישומים מסחריים ואינו רוצה לבזבז את זמנו בנסיונות לשנות את הסביבה בה הוא עובד (להיפך, הוא מעוניין שסביבת עבודתו לא תשתנה כל עוד הכל עובד כמו שצריך).

- אך כל הייתרונות הנ"ל הופכים לחסרונות גדולים עבור איש מחשב מיקצועי המעוניין לעצב וללמוד את סביבת העבודה שלו, לבחון אותה על ידי נסיונות, ולפתח אותה.
- דוגמא קנונית למערכת פתוחה היא מערכת ההפעלה Unix. מערכת זו נוצרה על ידי מתכנתים עבור מתכנתים ואנשי מקצוע למטרות מחקר ופיתוח ולכן הושקעו מאמצים גדולים לכך שהיא תהיה פתוחה ככל האפשר מבלי לפגוע ביעילותה ובביצועיה. היא נוצרה בתקופה בה המחשבים הקיימים היו גדולים, מועטים, ומאוד יקרים, אשר הגישה אליהם היתה רק לאנשי מקצוע מיומנים. לכן הנחת היסוד היתה שהמשתמשים הם מומחים או לפחות בעלי השכלה רחבה בתחום המחשבים ולכן הם יעדיפו מערכת גמישה ככל האפשר אותה יהיה ניתן להתאים בקלות לצרכים ולמטרות המיוחדות שלהם.
- טיעון: מערכות פתוחות מאפשרות את ההתפתחות של עצמן על ידי עצמן.
- קיימים היום גירסאות רבות של Unix עבור מחשבים אישיים, חלקן מסחריות (ויקרות) וחלקן לא-מסחריות. הבולטת ביניהן כיום היא מערכת ההפעלה Linux אשר פותחה בהתחלה על ידי סטודנט בגיל 21 ולאחר מכן על ידי מאות תכנתים נוספים ברחבי העולם באמצעות רשת האינטרנט. מאחר ושוק המחשבים האישיים מיועד ברובו עבור משתמשים שאינם

מתכנתים או בעלי השכלה במדעי המחשב, השימוש במערכת Linux עדיין מוגבל לאנשי מקצוע וסטודנטים בדרך כלל למטרות מחקר, פיתוח, והוראה. נעשים מאמצים גדולים ליצירת מימשקים נוחים וקלים למשתמש הפשוט. התהליך בעיצומו.

פרק 2

סקירה היסטורית



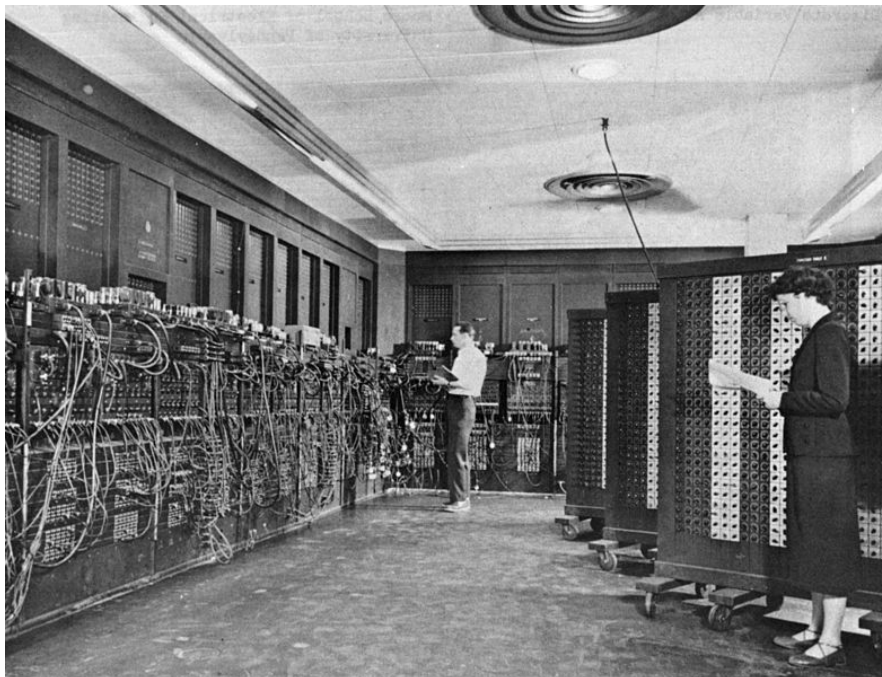
איור 2.1: Image courtesy of Computer History Museum, www.computerhistory.org

בכדי להבין מה זאת מערכת הפעלה יש להבין מה היו הבעיות הרבות שנתקלו בהם מערכות ההפעלה הישנות ("דינוזאורים") במהלך ההיסטוריה הקצרה יחסית של תחום המחשבים, וכיצד

פיתרון הוביל לפיתוחן של מערכות ההפעלה המודרניות כפי שאנו מכירים אותם היום.

2.1 הדור הראשון

- המחשבים הראשונים היו מאוד גדולים בנפחם (לפעמים התפרסו על פני מספר חדרי בניין), עלו מיליוני דולרים, ומאוד לא יעילים בהשוואה למחשבים הביתיים של היום. הם היו מורכבים מאלפי או עשרות אלפי שפופרות. לא היו שפות תיכנות, אפילו לא שפות סף! כל התוכניות נכתבו בשפת מכונה והיה צורך להזין פקודה אחר פקודה באופן ידני על ידי מתגים.



איור 2.2: Image courtesy of Computer History Museum, www.computerhistory.org

- בכדי להריץ תוכנית, על המתכנת להירשם לתור, ואז לחכות ליום ולשעה שנקבעו עבורו. בזמן שהוקצב לו היתה לו שליטה מלאה על כל המחשב. רק התוכנית שלו יכלה לרוץ על המחשב.

בזמן הריצה היה עליו להתפלל שאף אחת מ-20,000 השפופרות לא תישרף באמצע, וששום שגיאת תיכנות לא תיתגלה בתוכניתו. אם אירעה שגיאה במהלך התוכנית, המתכנת קיבל את הדפסת הפלט, כולל מצב הזיכרון, והרגיסטרים של המחשב. היה עליו לפנות את המחשב למתכנת הבא, לבדוק שוב את תוכניתו, ולהרשם שוב לתור.

- זיהוי שפופרת שנשרפה היה עשוי להימשך זמן רב, והתיחזוק השוטף של המחשב צרך בסביבות 10 טכנאים מיומנים במשרה מלאה.
- בזמן טעינת התוכנית או בהדפסת הקלט, יחידת העיבוד המרכזית של המחשב לא נוצלה. בהתחשב בעלות הגבוהה של המחשב (מאות אלפי או מיליוני דולרים), היה זה מצב בלתי נסבל. בכדי לפתור בצורה חלקית בעייה זו, הועסקו אנשי מיקצוע שהיו מיומנים בטעינה מהירה של תוכניות ובהוצאת הפלט הדרוש. החל מהשלב הזה, המתכנתים מסרו את תוכניותיהם להרצה למתחזק וקיבלו אותם מאוחר יותר.
- המצב השתפר קצת בתחילת שנות החמישים לאחר המצאת הכרטיסים המנוקבים.

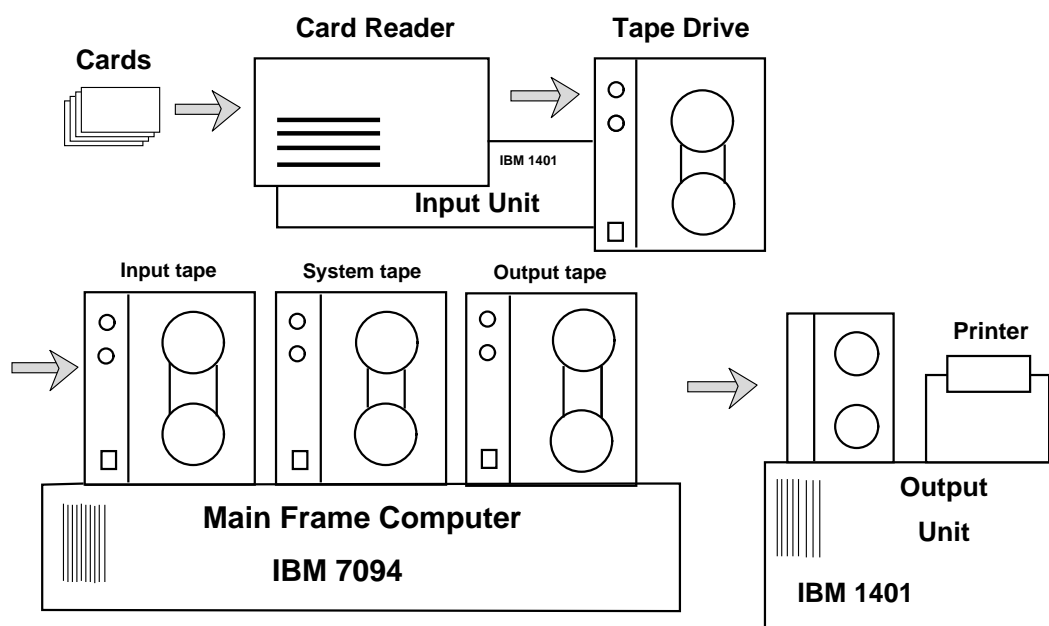
2.2 הדור השני 1955-1965



איור 2.3: Image courtesy of Computer History Museum, www.computerhistory.org

- המצאת הטרנזיסטור באמצע שנות החמישים שינתה את כל התמונה. הטרנזיסטור היה אמין הרבה יותר מהשפופרת, מה שהגביר את הייצור והמכירה של מחשבים.
- התקדמות זו איפשרה את הפיתוח של מערכת ההפעלה הפרימיטיבית הראשונה: Batch System. במקום להריץ תוכנית אחת בכל פעם, קבוצה של מספר תוכניות נטענו לתוך טייפ גדול (תהליך שנמשך מספר שעות) על ידי מחשב זול. לאחר מכן הטייפ הועבר למחשב המרכזי. האחראי טען קודם תוכנית מיוחדת ("מערכת ההפעלה") שתפקידה היה לקרוא תוכנית אחר תוכנית מתוך טייפ הקלט ולהריץ אותן אחת אחרי השניה.

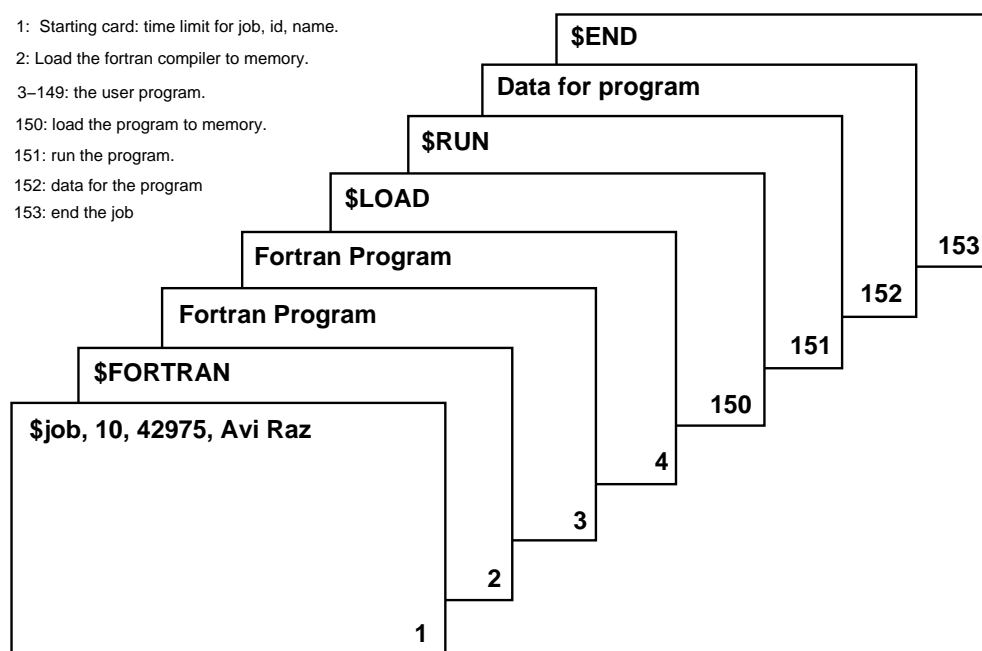
תוכנית זו היתה תוכנית קטנה שנקראה Resident Monitor משום שהיא ישבה באופן קבוע בזיכרון, וכל תפקידה היה לטעון את התוכניות שבטייפ הקלט, על פי סדר ההופעה בטייפ. הפלט נרשם בטייפ הפלט, שמאוחר יותר הועבר שוב למחשב זול לשם הדפסה על נייר. הפלט המודפס הועבר לחדר הפלט, אשר אליו המתכנתים היו באים בכדי לקבל אותו. בזמן טעינת הקלט או הדפסת הפלט, המחשב המרכזי היה פנוי לעבודה על קבוצה אחרת של משימות. התהליך מתואר בדיאגרמות הבאות:



איור 2.4: מערכת ההפעלה Batch System, 1955-1965

- דוגמא טיפוסית של תוכנית מחשב מאותם הימים: המתכנת היה כותב את תוכניתו על נייר רגיל עם עט או עפרון, ולאחר מכן מקליד אותה על מכונה מיוחדת להדפסתה על כרטיסים מנוקבים. את הכרטיסים המנוקבים היה עליו למסור לאחראי על המחשב.

- לשיטת ההפעלה Batch System היו כמובן ייתרונות גדולים על פני השיטה הישנה. נניח למשל שהאחראי קיבל תוכנית FOR-TRAN ממתכנת א', תוכנית COBOL ממתכנת ב', ותוכנית FORTRAN ממתכנת ג'. בשיטה הישנה היה עליו לטעון את הקומפיילר של FORTRAN לפני הרצת תוכנית א', לטעון את תוכנית א', ולהריץ אותה. לאחר מכן הוא צריך לעשות דבר דומה עם תוכנית ב' ועם הקומפיילר של COBOL, ולבסוף, בתוכנית ג', עליו שוב לטעון את הקומפיילר של FORTRAN וכו'. ברור שיש כאן ביזבוז מיותר של זמן ומשאבים.

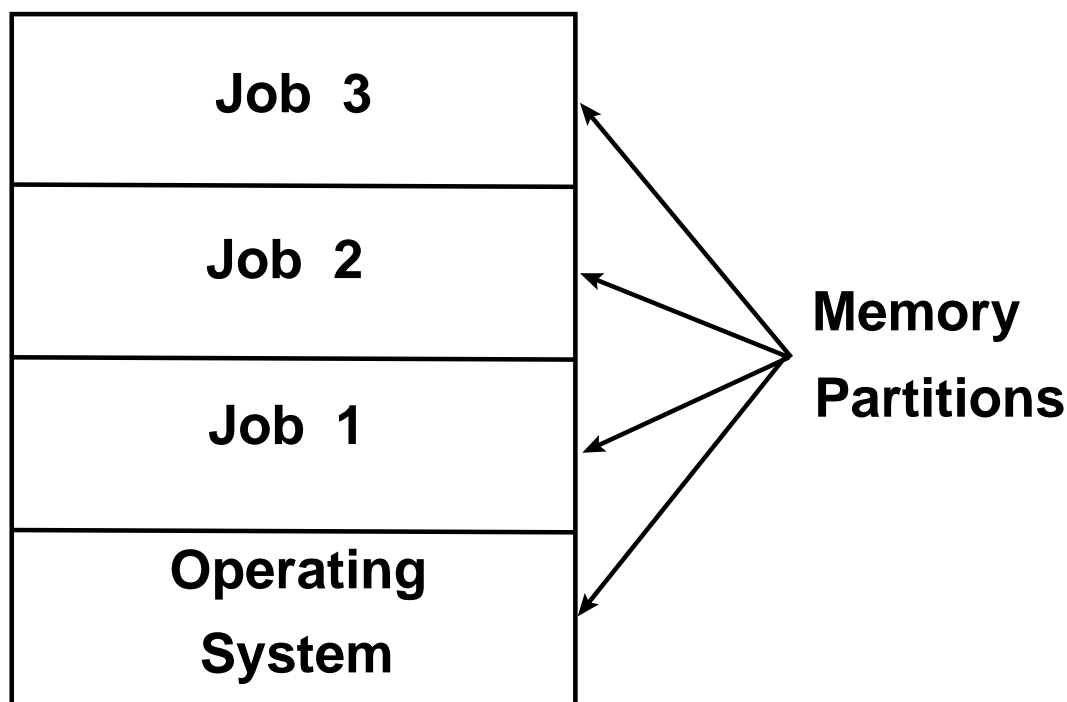


איור 2.5: תיכנות באמצעות כרטיסים מנוקבים

- בשיטה החדשה, האחראי של המחשב היה מקבץ את כל התוכניות של FORTRAN בקבוצה אחת. תידרש טעינה חד-פעמית של הקומפיילר של FORTRAN.

2.3 הדור השלישי 1965-1980

- למרות השיפורים הגדולים שחלו במעבר לשיטת ה resident monitor נותרו עוד כמה בעיות מעיקות. בזמן קליטת הקלט או בעיבוד נתונים מסחרי (בנקים ומוסדות ממשלה) פעולות טעינת קלט או פליטת פלט של עבודה מטייפים לטייפים ארכו לפעמים שעות רבות. בעבודות מסוג זה בין 80 ל-90 אחוז מהזמן התבזבז על קריאה וכתיבה לטייפים. בכל הזמן הזה, יחידת העיבוד המרכזית נותרה בבטלה מוחלטת.
- נעשו מאמצים גדולים מאוד בכדי לפתור את הבעייה הזו. חברת IBM פיתחה מערכת הפעלה בשם OS/360 עבור המחשבים הגדולים שלה, אשר לראשונה הכילה את אחד המרכיבים הכי חשובים של מערכת הפעלה מודרנית - multiprogramming.
- הרעיון היה לחלק את שטח הזיכרון למספר קטעים, קטע אחד לכל תוכנית. באופן כזה יהיה ניתן לטעון לזיכרון המחשב כמה תוכניות שונות (במקום תוכנית אחת בכל פעם). כמובן, בכל שלב רק תוכנית אחת תוכל לפעול, אך יהיה ניתן לעבור מתוכנית אחת לשניה במהירות בזק. בפרט, כאשר תוכנית אחת נעצרה בכדי לחכות לקריאה או כתיבה לטייפ, תוכנית אחרת תוכל לפעול, ועל ידי כך המעבד ינוצל בצורה טובה יותר.



איור 2.6: מערכת שיתוף זיכרון

- שיפורים נוספים באו לאחר כניסת הדיסקים הקשיחים לשוק המחשבים. נוצרה האפשרות לקרוא כרטיסים מנוקבים באופן ישיר לדיסק המחשב, תוך כדי דילוג על שלב הטייפ. בכל פעם שתוכנית אחת סיימה את עבודתה, מערכת ההפעלה טענה את התוכנית הבאה מתוך הדיסק לקטע הזיכרון שהתפנה. טכניקה זו נקראת *spooling*:

Simultaneous Peripheral Operation On Line

והיא פעלה באותה מידה גם עבור הפלט.

- המעבר לדיסק הקשיח סייע מאוד לרעיון ה-*multiprogramming* משום שמערכת ההפעלה יכלה לקרוא ולכתוב לאזורים שונים בדיסק באותו הזמן. למשל היתה קיימת האפשרות לטעון

תוכנית חדשה מתוך הדיסק ובאותו הזמן לכתוב את הקלט של תוכנית אחרת למקום אחר בדיסק. פעולה כזו היתה בלתי אפשרית עם טייפ.

Sequential-Access Device and Random-Access Device

2.4 מערכות שיתוף זמן (Time-Sharing Systems)

- לאחר כניסת הצגים והמקלדות לשוק המחשבים נסללה הדרך לרעיון ה-timesharing. (או ה-multitasking). זהו למעשה שכלול לוגי של רעיון ה-multiprogramming. במערכת שיתוף זמן, יחידת העיבוד המרכזית תוכל להפעיל מספר תוכניות לא רק אחת לאחר השניה, כפי שזה היה קודם, אלא גם אחת עם השניה, תוך כדי מעבר מתוכנית לתוכנית בפרקי זמן כל כך קצרים, שאף אחד ממשתמשי הקצה לא ירגיש בכך, ויחשוב כי יחידת העיבוד עסוקה רק עם התוכנית שלו.

- מערכות הפעלה מהסוג החדש כמובן צריכות להיות מורכבות יותר:

א. ניהול זיכרון וניהול תהליכים מורכב יותר

לפני שמערכת ההפעלה מפסיקה פעולת תוכנית אחת באמצע הריצה שלה ועוברת לתוכנית שניה, עליה "לצלם" ולשמור את כל הפרטים הדרושים לה בכדי לחזור יותר

מאוחר לתוכנית הראשונה ולהמשיך אותה בדיוק מהקודה בה היא נעצרה. כלומר, מערכת ההפעלה צריכה להכיל מנהל זיכרון ומנהל תהליכים הרבה יותר מורכב משהיה קיים קודם לכן.

ב. הגנה ובטיחות

מערכת ההפעלה צריכה להכיל מנגנונים המבטיחים שתוכניות שונות הרצות בו־זמנית לא יתערבו או יפריעו אחת לשניה. מילוי דרישה זו תלוי במידה רבה גם בחומרה. יש לייצר חומרה מתאימה אשר תתמוך בסוג כזה של מערכת הפעלה.

ג. מערכת קבצים

בכדי שהמשתמשים השונים יוכלו לשמור את התוכניות, הקלטים, והפלטים, דרושה **מערכת קבצים**. מערכת ההפעלה נדרשת לדעת לנהל היטב את מערכת הקבצים בכדי לשמור על סדר בדיסק. דרישה זו קשורה גם לדרישה הקודמת.

ד. מדיניות שיבוץ (scheduling)

כאשר כמה תוכניות הנמצאות בזיכרון מוכנות לפעולה (או להמשך פעולה לאחר הפסקה) באותו הזמן, על מערכת ההפעלה לבחור אחת מתוכן. החלטה זו צריכה להתבסס על אלגוריתם מתאים באופן כזה שזמן המעבד יתחלק ביניהן בצורה הוגנת ויעילה.

- על מחשב אחד יוכלו לעבוד מספר גדול של משתמשים בו זמנית. כל משתמש יוכל להריץ מספר תוכניות בו-זמנית. המשתמש יוכל להגיב בזמן אמת לאירועים חריגים העלולים להתרחש בזמן הריצה של תוכניותיו.
- מערכת שיתוף הזמן הרצינית הראשונה CTTS פותחה באוניברסיטת MIT בשנת 1962 אך זכתה להכרה ולפופולריות רק לאחר מספר שנים לאחר שפותחה חומרה מספיק טובה שתתמודד בה.
- לאור ההצלחה של המערכת CTTS, אוניברסיטת MIT, חברת הטלפונים Bell Labs, וחברת General Electrics התאגדו בכדי לפתח מערכת הפעלה אשר תוכל לספק שרותי מיחשוב למאות ואפילו לאלפי משתמשים. המודל שלהם היה המודל של חברת החשמל, והיה אפילו חזון לספק שרותי מיחשוב לכל אזרחי העיר בוסטון בדומה לאופן שבו מסופק חשמל.
- מערכת ההפעלה שתוכננה נקראה Multics (1965-1970)

Multiplexed Information and Computing Service

הושקעו מאמצים כבירים לשם השגת מטרה זו, אך ככל שעבר הזמן התברר כי העניין הוא הרבה יותר מסובך מכפי שזה היה נדמה בהתחלה. התוצאות לא היו משביעות רצון. לאחר השקעה כספית גדולה, חברת Bell Labs פרשה מהפרוייקט וחברת Gen-eral Electrics פרשה לחלוטין מענף המחשבים, ורוב האנשים

שעבדו על הפרוייקט נותרו ללא תעסוקה.

- למרות שמערכת ההפעלה Multics לא השיגה את המטרות הגדולות שייעדו לה, היא פעלה בסופו של דבר, אם כי לא כפי שתוכנן. אבל היא השפיעה בצורה משמעותית מאוד על כל מערכות ההפעלה שפותחו מאוחר יותר. אפשר לומר כי הפיתוח של Multics היה למעשה ניסוי גדול אשר הניב רעיונות חשובים להמשך פיתוח נושא מערכות ההפעלה.

- אחד החוקרים שעסקו בפרוייקט Multics בשם Ken Thompson ניסה לפשט את המערכת הזו (עבור משתמש יחיד - Unit), והצליח לפתח מערכת הפעלה פשוטה יותר בשם Unix אשר מילאה את מרבית הדרישות של מערכת שיתוף זמו בצורה אלגנטית מאוד ומשביעת רצון.

2.5 הדור הרביעי 1980-1995

הדור הרביעי מתאפיין בכניסתם המהירה של המחשבים האישיים, ולאחר מכן בקישוריות ביניהן באמצעות רשת האינטרנט.

- פיתוח מעגלי LSI (Large Scale Integration) איפשר את פיתוח המחשבים האישיים.

- המחשבים האישיים הראשונים לא היו שונים בעוצמתם מהמיני-מחשבים כגון PDP-11, אך הם היו קטנים בהרבה, ומה שחשוב

יותר, זולים בהרבה.

- עלותם הנמוכה והיכולות הגרפיות המעולות של המחשב האישי גרמו לתפוצה רחבה שלהם גם בקרב משתמשים ללא שום רקע בתיכנות או במדעי המחשב, ואשר גם לא היתה להם שום כוונה לרכוש ידע כזה. כל זה הוביל לתעשייה ענקית של ייצור תוכנות user-friendly. היה זה מעבר חד מאוד מהמערכת OS/360, למשל, אשר בכדי ללמוד כיצד להשתמש בה היה צורך לעבור על כמה ספרים מסובכים מאוד.
- מערכות ההפעלה השכיחות ביותר עבור המחשבים האישיים היו DOS ו-Unix. הגירסאות המוקדמות של DOS היו די פשוטות וחלשות, אך גירסאות מאוחרות יותר כללו אפשרויות מתקדמות יותר ויותר. חלק גדול מהן הועתק מ-Unix. מאוחר יותר, רעיונות נוספים של Unix יובאו למערכת ההפעלה Windows NT.
- מרבית המחשבים האישיים של IBM ותואמיהם השתמשו במערכת ההפעלה DOS, ורוב המחשבים האישיים האחרים הופעלו על ידי Unix, במיוחד אלה אשר המעבד שלהם היה מסוג RISC.
- הידע הרב שהצטבר במהלך ההיסטוריה של המחשבים הגדולים תרם רבות לפיתוח המעבדים הראשונים עבור המחשבים האישיים, אך למרות זאת, רוב הדגש הושם על הנוחות של

המשתמש היחיד המפעיל את המחשב, ופחות על הניצול היעיל של המעבד והציוד ההיקפי. למשל, המעבדים הראשונים של Intel לא היו מסוגלים להגן על מערכת ההפעלה מפני תוכניות משתמש, משום שהחומרה לא תמכה בזאת. רק לאחר הפיתוח של המעבדים 286, 386, ניתן היה ליבא מערכת הפעלה כמו Unix למעבדים אלה.

- עבודה במצב מוגן הפכה להיות הכרחית לאחר המעבר לרשתות של מחשבים אישיים, בהן היתה גישה למשתמשים שונים לקבצים אישיים של משתמשים אחרים (לפעמים גם דרך קווי טלפון). כך שלמרות שבהתחלה היה נדמה כי מערכות הפעלה של מחשבים גדולים היו מיותרות עבור מחשבים אישיים, התברר יותר ויותר כי גישה זו מופרכת לחלוטין.

- **Network Operating Systems and Distributed Operating Systems**

התפתחות חשובה נוספת שהחלה להתרחש באמצע שנות ה-80 היא הגידול המשמעותי ברשתות מחשבים של מחשבים אישיים שהופעלו על ידי **מערכות הפעלה לרשתות או מערכות הפעלה מבוזרות**.

- במערכת הפעלה לרשתות, המשתמשים מודעים לקיומם של המחשבים השונים הקיימים ברשת ומסוגלים להתקשר אליהם לשם העברת קבצים או הרצת תוכניות מרחוק.

- במערכת הפעלה מבוזרת, המשתמש אינו מודע לריבוי המעבדים, ונידמה לו כל הזמן כי הוא עומד מול מחשב אחד למרות שמערכת ההפעלה יכולה להריץ את תוכניותיו על מחשבים שונים (מסיבות של יעילות ופיזור נכון של תהליכים בין המעבדים השונים העומדים לרשותה).

פרק 3

המבנה של מערכת הפעלה

- המבנה של מערכת הפעלה נקבע על פי המטרות שהיא צריכה לשרת. קיים מספר לא קטן של מערכות הפעלה, והמבנה שלהן שונה מאחת לשניה. למרות זאת קיימים כמה עקרונות ומטרות משותפות שבהן נעסוק בפרק זה.
- מערכת הפעלה צריכה להיבחן מכמה נקודות תצפית:
 - א. השרותים אותם היא מספקת.
 - ב. המימשקים עבור המשתמש ועבור התוכניות.
 - ג. החלקים המרכיבים את המערכת והקשרים ביניהם.
- מערכות הפעלה מודרניות הן בדרך כלל מערכות גדולות ומורכבות, ולכן הדרך היחידה לתכנונם ופיתוחם היא באמצעות

חלקים קטנים יותר מהן הן מורכבות. בסעיפים הבאים נפרט את שמונת החלקים העיקריים שמהן מורכבת מערכת הפעלה מודרנית.

3.1 ניהול תהליכים

- תהליך הוא כל תוכנית שמתבצעת. התהליך מתחיל להתקיים מהרגע הראשון שבו התוכנית החלה לפעול ומסתיים לאחר ביצוע הפקודה האחרונה בתוכנית.
- אין לבלבל בין תוכנית מחשב לבין תהליך! למשל קיימת תוכנית בשם edit אך ניתן ליצור באמצעותה עשרה תהליכים שונים על ידי עריכת עשרה קבצים שונים בו־זמנית.
- כל תהליך זקוק למשאבים בכדי לבצע את משימתו כגון זמן CPU, זיכרון, קבצים, והתקני קלט/פלט כמו שטח דיסק ומדפסת.
- **מערכת הפעלה כספק שרותים:** הקצאת המשאבים היא בתחילת התהליך או תוך כדי זמן הריצה. במערכות שיתוף זמן, לתהליך אין גישה ישירה למשאבים אלה, ובמידה והוא זקוק להן, עליו לבקש אותם ממערכת ההפעלה בצורה מסודרת.
- לאחר השימוש במשאבים השונים, מערכת ההפעלה תבצע את כל פעולות הניקיון והשיחזור הדרושות.

- במערכת שיתוף-זמן יכולה להריץ מספר תהליכים בו-זמנית. כמובן, בכל זמן נתון, רק פקודה אחת של תהליך אחד יכולה להתבצע, אך המערכת יכולה לעבור מתהליך אחד לשני במהירות גדולה.
- קיימת הבחנה בין תהליכי מערכת לבין תהליכי משתמשים. תהליכי מערכת הם תהליכים שנוצרו על ידי מערכת ההפעלה שמטרתם לנהל את מערכת המחשב ולטפל בתהליכי המשתמשים. תהליכים אלה נשארים בדרך כלל באופן קבוע כל זמן שהמערכת פועלת.
- בכדי לנהל תהליכים, מערכת ההפעלה צריכה להיות מסוגלת לבצע את הפעולות הבאות:
 - א. יצירה או סיום של תהליכי מערכת ותהליכי משתמשים.
 - ב. השהיה או חידוש של תהליכים.
 - ג. IPC - Inter Process Communication
כלים המאפשרים תקשורת בין תהליכים.
 - ד. Inter Process Synchronization
כלים המאפשרים תאום זמנים בין תהליכים (סינכרון).
 - ה. כלים לשם יציאה ממצבי קיפאון (Deadlock).

3.2 ניהול זיכרון

- ניהול הזיכרון היא אחת המשימות הכי חשובות והכי מורכבות של כל מערכת הפעלה מודרנית.
- הפעולה השגרתית ביותר של יחידת העיבוד המרכזית היא קריאה וכתובה של בתים (bytes) ומילים מהזיכרון ואל הזיכרון. הזיכרון הראשי הוא בדרך כלל ההתקן היחיד אליו יחידת העיבוד מסוגלת לגשת באופן ישיר. למשל, בכדי שיחידת העיבוד תוכל לגשת לקובץ הנמצא בדיסק הקשיח, על הקובץ קודם כל להטען לזיכרון הראשי. טעינת הקובץ תבצע על ידי פקודה הנמצאת בזיכרון.
- בכדי שתוכנית כלשהי תבצע עליה להיטען לזיכרון. לאחר סיומה, שטח הזיכרון יתפנה לתוכנית חדשה.
- בכדי להגדיל את רמת השיתוף, היעילות, ומהירות התגובה, יש לאכלס בזיכרון מספר גדול ככל האפשר של תוכניות משתמשים.
- לשם ניהול הזיכרון על מערכת ההפעלה
 - א. לדעת בכל רגע נתון מהם חלקי הזיכרון התפוסים ועל ידי מי.
 - ב. להחליט איזה תוכנית (מתוך תור של תוכניות ממתונות) תעלה לזיכרון ברגע שמתפנה שטח מתאים בזיכרון. בדרך כלל קיים תהליך מערכת שנקרא scheduler אשר מטרתו

להחליט מה תהיה התוכנית הבאה שתעלה לזיכרון. קיים גם תהליך בשם swapper שמטרתו להוריד לדיסק באופן זמני תוכניות שרצו זמן מספיק בכדי שזמן ה-CPU יתחלק באופן הוגן בין כל התוכניות שצריכות לרוץ.

ג. להקצות או לפנות זיכרון באופן דינמי עבור תוכניות המעוניינות בזיכרון נוסף בזמן ריצתן.

3.3 ניהול התקני איחסון משניים

- מאחר והזיכרון הראשי הוא יקר ותלוי בהפעלת מתח לשם החזקת הנתונים עליו, הוא בדרך כלל קטן מאוד ביחס לכמות הרבה של הנתונים שעל מערכת מחשב להכיל. כל מערכת מחשב רצינית חייבת להכיל גם התקני איחסון משניים כגון דיסקים קשיחים, כונני טייפים, כונני CD-ROM וכו'.

- התקנים אלה מיועדים לשמירת תוכניות מערכת, תוכניות משתמשים, קבצים וטבלאות. מהירותם היא בדרך כלל איטית מאוד ביחס למהירות המעבד והזיכרון הראשי, ולכן אופן ניהולם הוא בעל חשיבות מרכזית עבור מערכת המחשב.

- מערכת ההפעלה חייבת להכיל כלים מתאימים עבור

א. ניהול השטחים הפנויים של התקני האחסון המשניים.

ב. הקצאת שטח על התקנים אלה עבור תהליכים הדורשים זאת.

ג. תיזמון - Disk Scheduling. דוגמא: תהליכים רבים מבקשים, בזמנים קרובים מאוד, לקרוא קבצים שונים הנמצאים באזורים שונים של הדיסק הקשית. אם מערכת ההפעלה תגיב באופן מיידית לכל בקשה כזו, יתבזבז זמן יקר על קריאת בלוקים שונים בדיסק. לפעמים רצוי לרכז את כל הבקשות האלה ביחד, ובזמן יותר מאוחר לבצע את כולן על פי סדר הבלוקים בדיסק בצורה יותר יעילה.

3.4 ניהול מערכת קלט/פלט

- אחת המטרות החשובות ביותר של מערכת ההפעלה היא להסתיר מפני המשתמשים את המורכבות העצומה של ההתקנים השונים של מערכת המחשב.
- במערכות מחשב ישנות, קריאת בלוק של נתונים מתוך דיסק מסוים הצריכה הכרה טובה של סוג הדיסק, וכתובת פקודה מורכבת מאוד, אשר רק מתכנת מיקצועי ומנוסה היה מסוגל לכתוב.
- במערכת ההפעלה UNIX, למשל, קיימת מערכת קלט מיוחדת (I/O system) שתפקידה להסתיר את המימשק להתקני הקלט/פלט אפילו ממערכת ההפעלה עצמה!

- מערכת הקלט/פלט חייבת להכיל

א. מערכת חציצה - Buffer-Caching System.

ב. מערכת למימשק בין דרייבר והתקן - Device-Driver Interface.

ג. דרייברים עבור ההתקנים הספציפיים של מערכת המחשב.

- דרייבר היא תוכנית מחשב המיועדת לספק שרותי קלט/פלט עבור התקן ספציפי. מערכת הדרייברים חייבת להיות עצמאית ומופרדת ככל האפשר בכדי שהתאמת מערכת ההפעלה למערכות מחשב שונות לגמרי תתאפשר בקלות.

3.5 ניהול מערכת קבצים

- מערכת מחשב עשויה להכיל מספר גדול של התקני איחסון מסוגים שונים. לכל התקן כזה יש המאפיינים המיוחדים לו, כגון יחידות איחסון פיזיות בגדלים ומאפיינים שונים sectors, blocks, tracks, clusters, שיטת כיתוב (addressing) מיוחדת ליחידות איחסון אלה, מהירויות גישה שונות, ואופן גישה (סדרתי או אקראי).
- כל הפרטים האלה אינם צריכים לעניין את המשתמש ולפעמים גם לא את מערכת ההפעלה. על מערכת ההפעלה לארגן את כל

המידע המאוחסן, בכל התקני האיחסון השונים, בצורה לוגית פשוטה ונוחה למשתמשים ולעצמה.

- יחידת האיחסון הלוגית המקובלת היא **הקובץ - file**.
- קבצים מאורגנים על ידי ספריות (directories)
- קובץ עשוי להיות מורכב ממספר גדול של בלוקים, סקטורים, או קלסטרים, הפרושים על פני מספר פלטות של דיסק או טייפ, בסדר אקראי לחלוטין (fragmented). עובדה זו אינה צריכה להיות ידועה או בכלל לעניין את המשתמש. למעשה, במרבית המקרים, עובדה זו אינה מעניינת גם את מערכת ההפעלה אשר ניגשת לקובץ באמצעות מנהל התקן (דרייבר) שעושה את העבודה עבורה.
- מערכת ההפעלה נגשת לקבצים וספריות באמצעות דרייברים מתאימים אשר תפקידם לגשת להתקנים המתאימים ולקרוא או לכתוב אליהם על פי בקשת המערכת. מערכת ההפעלה אינה צריכה להכיר את המבנה הספציפי של ההתקנים שבמערכת המחשב.
- מערכת ההפעלה צריכה להיות מסוגלת לבצע את הדברים הבאים:

א. יצירה או מחיקה של קבצים.

ב. יצירה או מחיקה של ספריות.

ג. להכיל כלים לשם פעולות על קבצים וספריות. למשל פקודות לשם הצגת רשימת קבצים בספריה מסוימת, העברת קובץ מספריה אחת לשניה, חיפוש קבצים, שינוי שמות של קבצים וספריות וכו'.

3.6 הגנה

- במערכת שיתוף זמן, תהליכים שונים עשויים להשתייך למשתמשים שונים. דבר דומה חל גם על קבצים, קטעי זיכרון, והתקנים. מערכת ההפעלה חייבת לדאוג לכך שהתהליכים השונים לא יפריעו אחד לשני, ולא יפעלו על קבצים, קטעי זיכרון, והתקנים שאינם ברשותם.
- לדוגמא, חומרה מתאימה תדאג לכך שתהליכים יוכלו לפעול רק בתוך מרחב הזיכרון המוקצה להם. כמו־כן, תהליך אחד לא יוכל להשתלט על כל זמן המעבד ולא יאפשר לתהליכים אחרים לפעול.
- לתהליכי משתמשים לא תהיה גישה ישירה להתקני קלט/פלט.

3.7 תיקשורת בין מחשבים שונים

- בניגוד לתחזיות מוקדמות באשר ליעודם של המחשבים האישיים, רובם כיום משתייכים לרשתות, ובכך הם הופכים להיות פחות

ופחות אישיים. מרבית המחשבים האישיים (והגדולים) מקושרים או מסוגלים להתקשר למחשבים אחרים בדרכים שונות ומגוונות.

- מרבית המחשבים האישיים במוסדות אקדמיים, מוסדות ממשלה, או חברות מסחריות מקושרים ביניהם על ידי רשת מחשבים מקומית

LAN - Local Area Network

וברוב המקרים, רשת זו מקושרת אל רשתות מחשבים מרוחקות יותר

WAN - Wide Area Network

באמצעות קווי טלפון אן קווי תיקשורת יותר מהירים.

- אפילו מחשבים ביתיים נמצאים בחלק מהזמן ברשת לאחר התחברות אליה באמצעות קווי טלפון.
- עובדה זו מאלצת כל מערכת הפעלה מודרנית לכלול כלים בכדי לאפשר לתהליכים על מחשבים שונים לתקשר ולפעול במשותף.
- אחת ההגדרות המקובלות של מערכת פתוחה היא: מערכת פתוחה היא מערכת המסוגלת להתקשר אל כל מערכת פתוחה אחרת על ידי ציות לכללים מוסכמים בנוגע לפורמט, תוכן, והמשמעות של ההודעות הנשלחות ומתקבלות (proto-cols). הגדרה מסודרת בפרוטרוט נקבעה על ידי ISO,

ISO - International Standards Organization

- ברשת מחשבים מקומית או רחבה, לתהליכים השונים הרצים על המחשבים השונים אין מרחב זיכרון משותף או שעון משותף. לכן האופן שבו תהליכים כאלה יוכלו לשתף פעולה ביניהם חייב להיות שונה לחלוטין מהאופן בו תהליכים הרצים על אותו מחשב משתפים פעולה.
- לכן מרכיב התיקשורת במערכת ההפעלה צריך להתמודד עם בעיות מורכבות מסוג אחר לגמרי. קיימות בעיות ניתוב (routing), בטיחות (security), גישה למשאבים משותפים, יציבות, ועוד.
- מצד שני, רשתות מחשבים תורמות באופן משמעותי ליעילותו ולמהירותו של כל מחשב ומחשב בודד, בכך שהם מגדילים את כמות המשאבים העומדים לרשותו. על ידי התחברות לרשת, מחשב בודד מגדיל בצורה משמעותית את כמות הקבצים והתוכניות העומדים לרשותו. במקרים רבים, המחשב הבודד יוכל גם להפעיל תוכניות על מעבדים נוספים, ולקבל שרותים אשר הוא עצמו אינו מסוגל לספק.

3.8 מעבד פקודות - Shell

- אחת מהתוכניות הכי חשובות של מערכת ההפעלה היא **מעבד הפקודות** או כפי שהיא נקראת לפעמים **מעטפת (shell)** או לפעמים גם **Command Interpreter**. תפקידה הוא לתווך

בין המשתמש השכיח ובין מערכת ההפעלה.

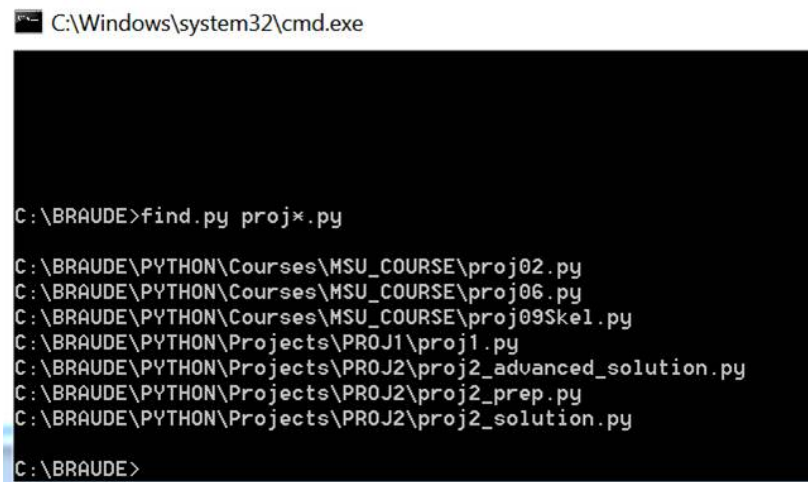
- תוכנית כזו דרושה עבור רוב המשתמשים מאחר והגישה הישירה למערכת ההפעלה היא מסובכת מאוד ומתאימה רק למשתמשים מקצועיים המכירים את המערכת לפרטיה ויודעים לכתוב תוכניות בשפת התיכנות בה היא כתובה (מה שהוא בלתי מציאותי עבור המשתמש הרגיל).
- לאותה מערכת הפעלה עשויים להיות כמה מעבדי פקודות המותאמים לטעמים ומטרותיהם של המשתמשים השונים.
- מערכות הפעלה מסוימות כוללות מעבד פקודות בתוך הגרעין. מערכות הפעלה אחרות, כמו Unix או Dos, מתייחסות למעבד פקודות כאל תוכנית מיוחדת, שמתחילה לפעול ברגע שהמשתמש נכנס למערכת, או כאשר המחשב מאותחל. במערכת Unix, כל משתמש מקבל מעבד פקודות אישי. לכן אם בזמן נתון מספר המשתמשים על המחשב הוא 100, מספר מעבדי הפקודות יהיה 100 לפחות (משתמש מסוים יכול להשתמש בכמה מעבדי פקודות בו־זמנית).
- מעבד הפקודות הסטנדרטי של Dos הוא Command.com. מעבדי הפקודות הסטנדרטיים של Unix הם
 - א. Bourne Shell - sh
 - ב. C Shell - csh
 - ג. Turbo C Shell - tcsh

ד. Korn Shell - **ksh**

ה. Z Shell - **zsh**

ו. Bourne Again Shell - **bash**

המשתמש יכול לבחור את מעבד הפקודות המועדף עליו בכל זמן. ניתן גם להפעיל מעבד פקודות באופן זמני בכל זמן שהמשתמש חפץ בכך.



```

C:\Windows\system32\cmd.exe

C:\BRAUDE>find.py proj*.py

C:\BRAUDE\PYTHON\Courses\MSU_COURSE\proj02.py
C:\BRAUDE\PYTHON\Courses\MSU_COURSE\proj06.py
C:\BRAUDE\PYTHON\Courses\MSU_COURSE\proj09Skel.py
C:\BRAUDE\PYTHON\Projects\PROJ1\proj1.py
C:\BRAUDE\PYTHON\Projects\PROJ2\proj2_advanced_solution.py
C:\BRAUDE\PYTHON\Projects\PROJ2\proj2_prep.py
C:\BRAUDE\PYTHON\Projects\PROJ2\proj2_solution.py

C:\BRAUDE>
  
```

איור 3.1: מערכת ההפעלה DOS

- מעבד פקודות מספק למשתמש מימשק פשוט ונוח לשרותים השונים של מערכת ההפעלה. ההבדלים בין מעבד פקודות אחד לשני הוא בדרך כלל ברמת היכולת לנצל את שרותי המערכת בצורה המירבית. מעבדי פקודות מסוימים הם יותר ידידותיים למשתמש מאחרים.

- למרות זאת שום מעבד פקודות אינו מספיק חזק בכדי לנצל את המערכת בצורה המקסימלית. הדרך היחידה לעשות זאת היא על ידי כתיבת תוכניות בשפת התיכנות של המערכת, ולפעמים

גם בשפת סף.

```

samy@linux-14xm:~> ls -l /usr/bin | wc -l
2591
samy@linux-14xm:~> cd shared
samy@linux-14xm:~/shared> ls -l
total 21
-rwxrwxrwx 1 samy users 7540 Mar 29 00:34 basics.py
-rwxrwxrwx 1 samy users 7605 Apr 12 18:44 basics.pyc
drwxrwxrwx 1 samy users 0 Apr 16 14:22 bkup
-rwxrwxrwx 1 samy users 1034 Apr 13 13:43 clipboard.txt
drwxrwxrwx 1 samy users 0 Apr 12 14:57 CODE
drwxrwxrwx 1 samy users 0 Apr 12 18:35 junk
-rwxrwxrwx 1 samy users 174 Apr 12 17:10 linux_share.txt
-rwxrwxrwx 1 samy users 1274 Apr 13 12:53 primes.py
-rwxrwxrwx 1 samy users 1990 Apr 12 19:31 primes.py.orig
drwxrwxrwx 1 samy users 0 Apr 12 18:07 tel
samy@linux-14xm:~/shared>

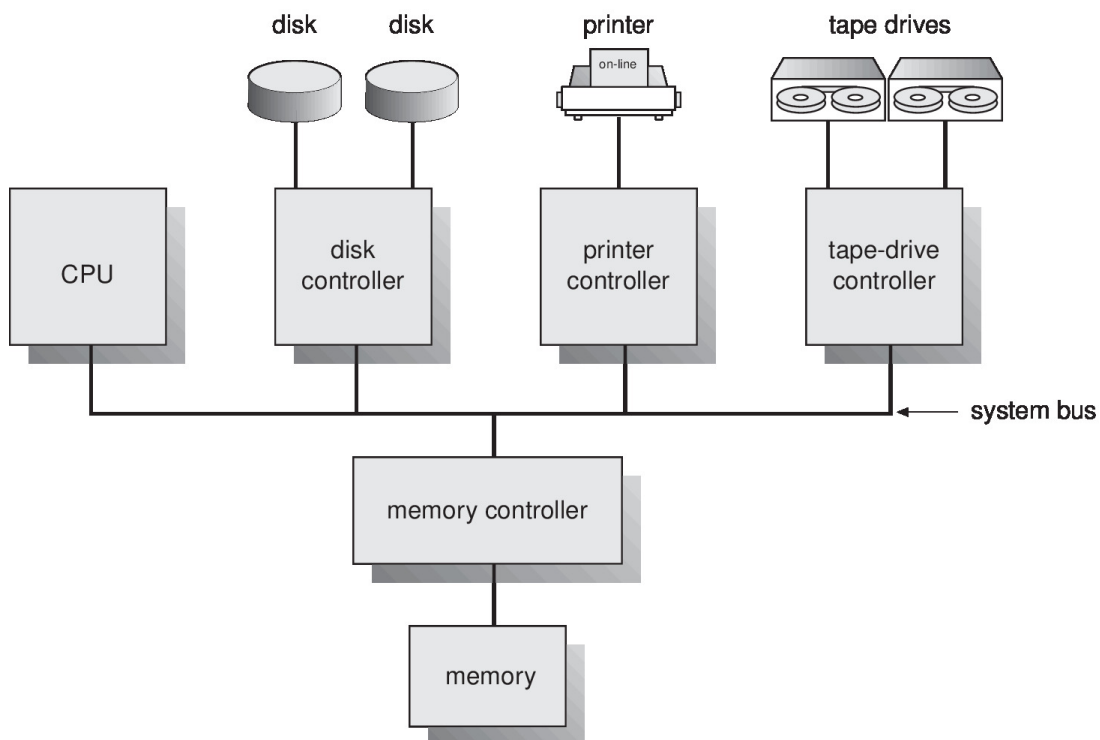
```

איור 3.2: מערכת ההפעלה Linux

- לאחר המעבר לצגים בעלי יכולת גרפית גבוהה, פותחו מעבדי פקודות (או מימשקים למעבדי פקודות ישנים) גרפיים ידידותיים במיוחד המאפשרים למשתמש לקבל את שרותי המערכת באמצעות תפריטים רבים ומגוונים ועל ידי שימוש בעכבר. דוגמא בולטת במיוחד היא מערכת ההפעלה של מחשבי Macin-tosh. למרות התפתחות זאת, משתמשים ותיקים ואנשי מקצוע עדיין מעדיפים את מעבדי הפקודות מבוססי מקלדת המאפשרים להם לבצע פעולות מסובכות יותר בקלות ובמהירות.
- מעבדי הפקודות מבוססי המקלדת כוללים גם לרוב שפת תיכנות פשוטה לשם כתיבת סקריפטים לשם ביצוע פעולות מורכבות במיוחד.
- מעבדי פקודות משמשים בעיקר לשם איתחול תהליכים, הפעלת יישומים, טיפול בקבצים, פעולות קלט/פלט, הגנה, ותיקשורת.

פרק 4

מבנה של מערכת מחשב



איור 4.1: Image courtesy of Operating System Concepts book, Silberschatz/Galvin/Gagne

4.1 פעולה שגרתית של מחשב

- מערכת מחשב מודרנית מורכבת מיחידת עיבוד מרכזית (CPU) ומספר בקרי התקנים (device controllers) המקושרים ביחד

על ידי ערוץ (system bus) המספק להם גישה ליחידת זיכרון משותפת.

- יחידת העיבוד והבקרים מסוגלים לפעול בו-זמנית.
- לכל הבקרים יש גישה ליחידת הזיכרון הראשי ולכן תיתכן תחרות ביניהם על הגישה אליה בזמן מסוים. עובדה זו מחייבת **בקר זיכרון** שתפקידו להבטיח גישה מסודרת של כל הבקרים האחרים לזיכרון.
- לשם הפעלת מחשב דרושה תוכנית איתחול bootstrap. זוהי בדרך כלל תוכנית קצרה מאוד שתפקידה להכין את כל התקני המחשב ולטעון את מערכת ההפעלה. תוכנית האיתחול חייבת לדעת איך למצוא את גרעין מערכת ההפעלה, לטעון אותו לזיכרון, ולהעביר אליו את השליטה במחשב.
- הגרעין מפעיל את התהליך הראשון (ב-Unix הוא נקרא init), ומחכה להתרחשות אירועים.
- התרחשות אירועים כלשהם במערכת המחשב מתבטאת לרוב על ידי פסיקות (interrupts) חומרה או תוכנה המגיעות ליחידת העיבוד.
- **פסיקות חומרה** מגיעות לרוב באמצעות הערוץ (system bus) מהבקרים השונים של המערכת. **פסיקות תוכנה** נגרמות לרוב על ידי הפעלת **קריאות שרות** (system calls) של מערכת ההפעלה.

- סוגי האירועים שיוצרים פסיקות חומרה הוא מגוון. למשל: בקר דיסק אשר סיים פעולת קלט או פלט, פעולת מקלדת, פעולת חילוק באפס, גישה לא חוקית לזיכרון, או הפעלת קריאת שרות של מערכת ההפעלה. לכל ארוע כזה קיימת שיגרה מתאימה שצריכה לענות למתבקש.
- כאשר המעבד מקבל פסיקה, הוא יפסיק לחלוטין את כל מה שעשה עד לאותו הרגע, ובהתאם לסוג הפסיקה הוא יעבור למקום מסוים בזיכרון המכיל כתובת של שיגרת פסיקה מתאימה. יחידת העיבוד תעביר את השליטה לשיגרה, ולאחר שהשיגרה תסיים, היא תחזור למה שעשתה קודם לכן, בדיוק מאותה הנקודה שבה הפסיקה.
- פסיקות הן חלק בלתי נפרד מכל מערכת מחשב מודרנית ובלעדיהן לא ניתן לתכנן ולפתח שום מערכת תוכנה רצינית (כמו מערכת הפעלה למשל). לכן חשוב מאוד להיות מודע לקיומן, להכיר אותן, ולעשות בהן שימוש מושכל. לכל מחשב יש מערכת פסיקות המיוחדת לו, אך קיימות בכל זאת מספר פסיקות המשותפות לכל המחשבים המודרניים.
- פסיקות נגרמות כתוצאה מאירועים מיוחדים הדורשים תשומת לב מיידית, לכן התגובה לפסיקה חייבת להיות מהירה ככל האפשר. המנגנון המקובל הוא להכין מראש (בזמן האיתחול למשל) טבלת פסיקות (interrupt vector) המכילה את כל

הכתובות של כל שיגרות הפסיקה לכל הפסיקות האפשריות במערכת. טבלה כזו שמורה בדרך כלל בחלק הכי נמוך של הזיכרון הראשי. כל שורה בטבלה כזו מכילה מספר מזהה עבור סוג הפסיקה (לכל התקן מותאם מספר כזה) וכתובת עבור השיגרה המתאימה. מערכות הפעלה כמו Dos ו-Untix משתמשות בשיטה זו.

- פרטי ביצוע הפסיקה שונים ממערכת למערכת. הדרך המקובלת היא לשמור את תוכן הרגיסטר PC במחסנית. אם שיגרת הפסיקה צריכה לשנות את מצב הרגיסטרים של המעבד, היא תהיה אחראית לשמירתם ולהחזרת המצב לקדמותו לכשתסיים.
- במחשבים רגילים, פסיקות נוספות יידחו בזמן הביצוע של פסיקה קודמת. מעבדים מתוחכמים יותר עשויים לקבל פסיקות נוספות בזמן הביצוע של פסיקה. במחשבים מסוג זה לכל פסיקה יש סדר עדיפות, והיא תוכל להפריע לפסיקה אחרת רק אם יש לה סדר עדיפות גבוה יותר.
- מחשבים מסוימים (למשל מחשבים אישיים תואמי IBM) מכילים לצד המעבד גם בקר פסיקות שתפקידו היחיד הוא לטפל בפסיקות השונות (ועל ידי כך להקל על יחידת העיבוד).

4.2 קלט ופלט

- התקני קלט ופלט מנוהלים לרוב על ידי בקרים מתאימים. על ידי כך נחסכת עבודה רבה מהמעבד אשר רצוי שיתעסק בדברים יותר חשובים.
- לכל התקן יש בקר משלו אך במערכות מחשב מתוחכמות יותר ייתכן בקר אחד לכמה התקנים. למשל בקר SCSI יחיד עשוי לנהל 8 התקנים בעת או בעונה אחת, ובמקרים רבים ניתן להוסיף לו התקנים חדשים ללא בעיות.
- לכל בקר יש מחיצת זיכרון מקומית משלו וכמה רגיסטרים שבאמצעותם הוא מתקשר עם יחידת העיבוד (או עם בקרים אחרים). גודל הזיכרון המקומי תלוי בהתקן. למשל בקר דיסק קשיח חייב להכיל זיכרון מקומי בגודל של סקטור אחד או כמה סקטורים. ברוב הזמן הבקר עסוק בהעברת נתונים מההתקן אותו הוא מנהל לזיכרון המקומי שלו הלוח ושוב.
- **ביצוע פעולת קלט/פלט:** המעבד טוען את הרגיסטרים של הבקר בפקודת קלט/פלט מסוימת (מדובר בשפת המכונה של הבקר עצמו!) וממשיך בעיסוקיו. הבקר קורא את הרגיסטרים שלו. אם למשל הוא מוצא פקודת קריאה מהדיסק, הוא יפנה לדיסק, יעביר את הנתונים למחיצה המקומית שלו, ובסיום הפעולה יעביר הודעה מתאימה למעבד באמצעות פסיקה מתאימה.

- לאחר בקשת הקלט/פלט, בפני המעבד עומדים שני סוגי פעולה אפשריים.

א. לחכות עד מילוי הבקשה: Synchronous I/O

ב. להמשיך מייד בתוכנית הנוכחית (אם זה אפשרי) או לשרת

תוכניות אחרות בתור: Asynchronous I/O

- באפשרות הראשונה יש שתי שיטות לחכות:
 - פקודת מעבד מיוחדת wait המשביתה את המעבד עד לפסיקה מהבקר.
 - הפעלת לולאה סתמית כמו: Loop: jmp Loop. המעבד יצא מהלולאה לאחר קבלת הפסיקה מהבקר.
- הלולאה שבאפשרות השניה כוללת לפעמים גם חיפוש שנקרא polling עבור התקני חומרה אשר אינם תומכים בשיטת הפסיקות המקובלת ובמקום ליצור פסיקות רגילות הם מדליקים רגיסטר מסוים בבקר שלהם ומחכים שמערכת ההפעלה תבחין בהם על ידי קריאת הרגיסטר הזה.
- האפשרות הראשונה עדיפה בדרך כלל משום שבאפשרות השניה פעולת הלולאה הסתמית מצריכה גישה סתמית לזיכרון ובכך מאטה את קצב הגישה של בקרים אחרים לזיכרון ומעסיקה את המעבד ובכך מונעת מתהליכים אחרים לרוץ.

- גם במערכת asynchronous I/O כמו Unix קיימת פקודת מערכת בשם wait אם התהליך מעוניין לחכות. למטרה זו, על מערכת ההפעלה לתחזק טבלת התקנים (device-status table).
- כל שורה בטבלה תכיל: סוג התקן, כתובת, מצב (מושבת, לא פעיל, פעיל).
- במידה וההתקן עסוק בבקשה מסוימת, הטבלה תכיל גם את פרטי הבקשה.
- בנוסף לכך, ניתן לכלול בטבלה גם תור של בקשות עבור ההתקן.
- כמובן, כל שיפור כזה מעמיס על מערכת ההפעלה תפקידים נוספים לשם ניהול התורים ועידכון הטבלה מידי פעם.

4.3 DMA - Direct Memory Access

- ריבוי התקני קלט/פלט והמספר הגדול של הפסיקות שנוצר על ידיהם מכביד מאוד על המעבד ומאט את קצב פעולתו. המעבד מבזבז זמן יקר בהעברת נתונים מהזיכרון לבקרי ההתקנים הלוח ושוב. דוגמא בולטת לכך היא המקלדת: כל תו שמוקלד עשוי ליצור פסיקה. בכדי להתגבר על בעייה זו, כמעט כל מחשב כיום כולל בתוכו רכיב DMA אשר תפקידו לשחרר את המעבד

הראשי מהמעמסה של העברת/קריאת נתונים מהזיכרון הראשי לבקרי ההתקנים.

● לשם העברת נתונים לדיסק, למשל, מערכת ההפעלה תמסור לרכיב DMA את ההוראות הבאות:

א. כתובת התחלה בזיכרון ומספר הבתים שיש להעביר
ב. מספר הבלוק הראשון בדיסק אליו יש לבצע את ההעברה
ג. צורת עבודה: שוטפת או בית אחר בית (עם החזרת שליטה למעבד לאחר העברת כל בית ובית)

לאחר העברת ההוראות האלה, יחידת העיבוד מתפנה מייד לעיסוקים יותר חשובים, ורכיב ה-DMA מבצע את עבודת ההעברה.

● בכדי לקרוא נתונים מהדיסק לזיכרון, מערכת ההפעלה תבצע את הפעולות הבאות:

א. הכנת מחיצה (buffer) מתאימה בזיכרון הראשי.
ב. לאחר הכנת המחיצה, המעבד יעביר לבקר ה-DMA את הכתובות והפרמטרים הדרושים לשם העברת הנתונים בדיסק למחיצה בזיכרון.
ג. רכיב ה-DMA יופעל על ידי איתחול סיבי בקרה של רגיסטרים הנמצאים עליו. יחידת העיבוד מתפנה לעיסוקים אחרים.

בקר ה-DMA יצור פסיקה לאחר סיום עבודתו.

- מחשבים תואמי IBM עם מעבד 80286 ומעלה מכילים שני רכיבי DMA. בנוסף לכך, מחשבים כאלה משתמשים בטכניקה מיוחדת memory mapped I/O. חלק נכבד של הזיכרון הראשי (בדרך כלל האזור 640K-1024K) שמור למערכת המחשב בלבד. חלק זה של הזיכרון ממופה באופן פיזי לרגיסטרים ולמחיצות של ההתקנים, כך שהגישה אליהם תתבצע ישירות דרך הזיכרון הראשי ולא דרך הבקרים. למשל לכל פיקסל של המסך מתאים תא בזיכרון, ולכן כתיבה למסך תתבצע על ידי כתיבה ישירה לזיכרון, מה שמשפר מאוד את מהירות התגובה של המסך.

4.4 מבנה האיחסון

- **זיכרון ראשי** - התקן האיחסון היחיד שיש למעבד גישה ישירה אליו.
- **התקני איחסון משניים** - הרחבה זולה ועמידה של הזיכרון הראשי.
- **דיסקים מגנטיים** - מתכת או זכוכית מצופה בשכבת חומר מגנטי. - משטח הדיסק מחולק למסלולים (tracks), וכל מסלול מחולק לגזרות (sectors).

- התיקשורת בין המחשב לדיסק מתבצעת בצורה לוגית מסודרת באמצעות בקר הדיסק.

● מערכת התקני איחסון נבחנת על פי אמות המידה הבאות:

- מהירות

- עלות

- עמידות

● **Caching**: העתקת נתונים להתקן מהיר יותר.

- אפשר לראות את הזיכרון הראשי כזיכרון cache עבור הדיסקים: כל מערכת הפעלה מודרנית מנהלת מערכת קבצים בזיכרון (File Cache) הראשי בנוסף למערכת הקבצים הרגילה, שהיא לרוב שיכפול מדויק של חלק מהקבצים בדיסק שנמצאים בשימוש באותו הזמן.

- טכניקת cache למעבד: 80%-90% פגיעה. כלומר, ברוב המקרים, כשתהליך מסוים ירצה לגשת לקובץ מסוים הוא כבר ימצא אותו בזיכרון, ובכך יחסוך זמן יקר ומשאבים (וגם שחיקה של הדיסק).

● למעשה טכניקת ה-Caching מתקיימת בכמה רמות:

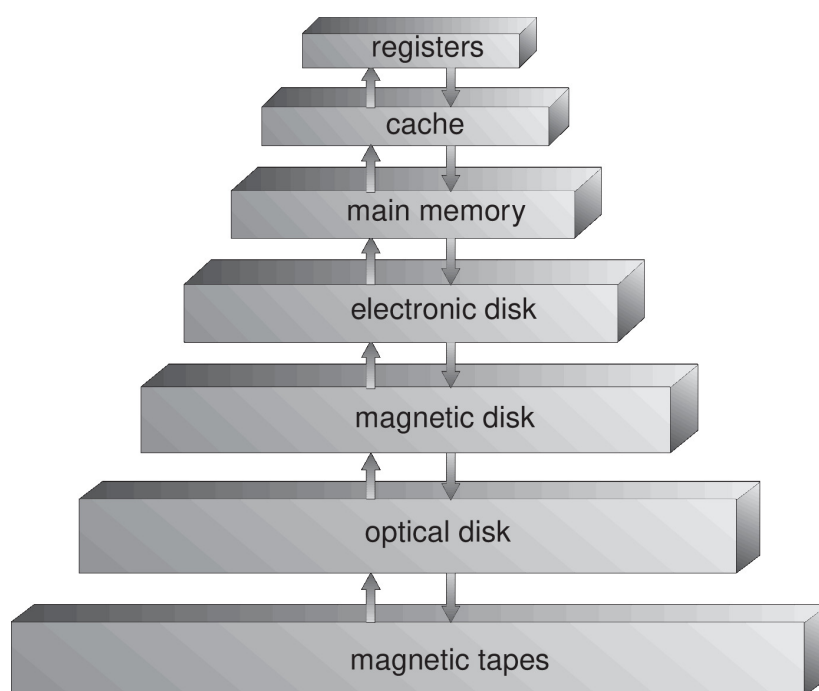
- ברמה הראשונה היא מתקיימת בין הדיסק לזיכרון הראשי

- ברמה השנייה היא מתקיימת בין הזיכרון הראשי ליחידת ה-L2 Cache שנמצא במעבד (יחידה כזו קיימת בכל ליבה נפרדת).

- ברמה השלישית היא מתקיימת בין יחידת ה-L2 Cache ליחידת ה-L1 Cache שנמצא במעבד וקרובה יותר לרגיסטרים.

- במעבדים מיוחדים (כמו Itanium או Alpha) קיימת גם רמה רביעית בין יחידת ה-L3 Cache ויחידת ה-L2 Cache. אך לרוב, יחידת ה-L3 Cache ממוקמת מחוץ למעבד ומשותפת לכל הליבות.

● סולם התקני איחסון



איור 4.2: Image courtesy of Operating System Concepts book, Silberschatz/Galvin/Gagne

● העתקת נתונים משלב אחד לשלב הבא בסולם היא לפעמים סמויה ולפעמים גלויה בהתאם לסוג החומרה ומערכת ההפעלה.

- העברת נתונים מהזיכרון הראשי לזיכרון cache של המעבד וממנו לרגיסטרים היא לרוב פעולה אוטומטית שמתבצעת

בחומרה באופן בלתי תלוי במערכת ההפעלה (מערכת ההפעלה אינה יכולה לשלוט במצב זה).

- מצד שני, העתקת נתונים מהדיסק לזיכרון (או מטייפ לדיסק) היא פונקציה של מערכת ההפעלה.

● אחידות ועקביות (cache coherency problem)

- ריבוי רמות cache יוצר בעיות ברמת החומרה וברמת מערכת ההפעלה. נתון מסוים או קובץ מסוים עשוי להופיע בכמה רמות של סולם ה-cache.

- שינויים שמתבצעים ברמה אחת חייבים להשתקף בכל הרמות האחרות במוקדם או במאוחר בכדי למנוע מתהליכים שונים לגשת לעותק לא מעודכן של הנתונים.

- בעייה כזו אינה קיימת במערכת הפעלה שאינה תומכת בריבוי תהליכים. במערכת מרובת תהליכים, מספר תהליכים עשויים לגשת לאותו הקובץ. יש להבטיח גישה לגירסה הכי מעודכנת של הקובץ.

- הבעיה הופכת ליותר מסובכת במחשבים מרובי מעבדים, ומסתבכת עוד יותר במערכות מבוזרות.

4.5 הגנה ברמת החומרה

- המעבר למערכת הפעלה מרובת תהליכים ומרובת משתמשים שיפר בצורה רצינית את הביצועים ואת רמת הניצול של המחשב, אך באותה מידה יצר גם בעיות קשות.
- במערכות ההפעלה הראשונות (multiprogramming), תוכנית מסוימת היתה עשויה לשנות תוכניות אחרות הנמצאות באותו הזמן בזיכרון, או לשנות אפילו את גרעין מערכת ההפעלה (שנמצא כל הזמן בזיכרון). נזק דומה יכל להיגרם גם למערכת הקבצים בדיסק. בעיות מסוג זה הן שכיחות במערכת ההפעלה Dos שאינה מוגנת.
- מצב זה חייב תיכנון מחדש לא רק של מערכת ההפעלה אלא גם של חומרת המחשב. חלק גדול של שגיאות התיכנות שעשויות לגרום לבעיות אלה מאובחן כבר ברמת החומרה.
- נסיון להפעיל פקודה בעלת תחביר שגוי או לגשת לכתובת שאינה במרחב הזיכרון של המשתמש תגרום מייד לפסיקה מתאימה ותעביר את השליטה למערכת ההפעלה (על ידי שימוש בוקטור פסיקות).
- מערכת ההפעלה עשויה לסיים את התהליך, ולזרוק את מצב הזיכרון של התהליך לקובץ בדיסק.
- ברמת החומרה נקבע כי יש להבחין בין מצבים שונים של עבודת

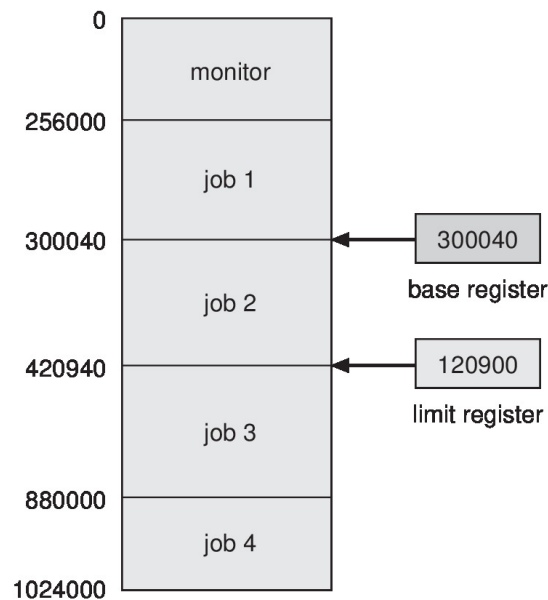
המחשב. הבחנה מינימלית היא בין **מצב משתמש** (user mode) ו**מצב מערכת** (system mode או monitor mode). הבחנה זו מיושמת על ידי הוספת סיבית מיוחדת (mode bit) לחומרת המחשב.

- סיבית המצב מאפשרת לנו לדעת אם המעבד מפעיל תהליך מערכת או תהליך של משתמש. כמובן, למשתמשים רגילים לא תהיה גישה ישירה לסיבית המצב.
- ברמת החומרה, קיימת הבחנה בין פקודות מכונה מוגנות לפקודות רגילות. החומרה תרשה הפעלת פקודות מוגנות רק במצב מערכת. נסיון להפעיל פקודות מוגנות במצב משתמש יגרום לפסיקה.
- מערכת ההפעלה Dos תוכננה עבור המעבד 8088 של אינטל שלא היתה בו סיבית מצב. מעבדים יותר מתקדמים של אינטל כוללים בתוכם סיבית מצב, וכתוצאה מכך מערכות הפעלה כמו WinNT או OS/2 שתוכננו למעבדים אלה מספקים שרותי הגנה טובים בהרבה מבעבר.
- תהליכי משתמשים עשויים לגרום נזק באמצעות פקודות I/O. לכן כל פקודות I/O יוגדרו כפקודות מוגנות. משתמש רגיל לא יוכל לבצען בצורה ישירה. כל בקשה לפקודת I/O תופנה למערכת ההפעלה. מערכת ההפעלה תבדוק אם למשתמש יש את ההרשאות המתאימות ותפעל בהתאם.

4.6 ניהול הגנת זיכרון

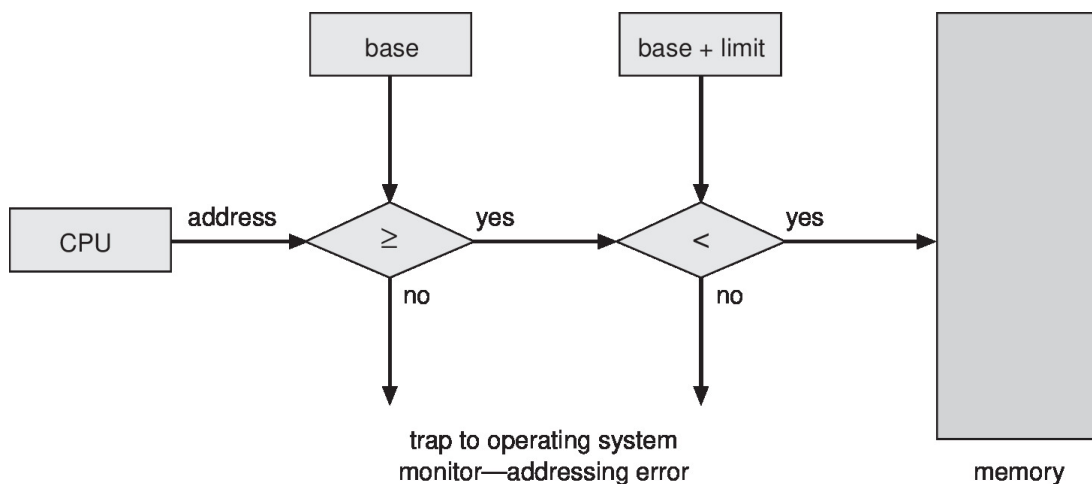
- תנאי הכרחי לפעולתו התקינה של המחשב הוא הגנה על וקטור הפסיקות ועל תוכניות השרות של מערכת ההפעלה מפני תהליכי המשתמשים.
- באופן כללי, לתוכניות משתמשים לא תהיה גישה ישירה למערכת ההפעלה.
- בנוסף לכך יש להגן על תהליכי המשתמשים שלא יפריעו אחד לשני. למשל, אסור לתהליך אחד לפלוש למרחב הזכרון של תהליך שני (גם לא עבור קריאה!).
- הגנה כזו חייבת להיות מיושמת ברמת החומרה.
- הגנה כזו מיושמת בכמה דרכים שונות (שנלמדות בקורס מתקדם יותר במערכות הפעלה).
- נציג רק מימוש אפשרי אחד שהוא שכיח מאוד: לכל תהליך יהיה מרחב כתובות בזיכרון אליו הוא רשאי לפנות. לשם כך דרושים שני רגיסטרים שיכילו שני ערכים `base` ו-`limit`. הרגיסטר הראשון יכיל את הכתובת הכי קטנה במרחב, והשני את הגודל המקסימלי של כתובת במרחב זה.
- ההגנה מושגת באמצעות חומרת CPU המשווה כל כתובת זיכרון שנוצרת במצב משתמש עם הרגיסטרים `base` ו-`limit`. אם יש

חריגה ממרחב הכתובות החוקי, תוצר פסיקה והתוכנית תופסק (segmentation fault).



איור 4.3: Image courtesy of Operating System Concepts book, Silberschatz/Galvin/Gagne

- רק מערכת ההפעלה רשאית לגשת ולטעון את שני הרגיסטרים $base$, $limit$ ולמטרה זו משתמשת בפקודות מכונה מוגנות המיועדות למטרה זו בלבד!



איור 4.4: Image courtesy of Operating System Concepts book, Silberschatz/Galvin/Gagne

- לרגיסטרים אלה אין שום משמעות במצב מערכת: הליך זה כמובן אינו חל על תהליכי מערכת ההפעלה אשר רשאית לגשת לכל כתובת בזיכרון ללא שום הגבלה. רק כך תוכל מערכת ההפעלה לטעון תוכניות משתמשים, להפסיק תהליכים, להעתיק או לפנות קטעי זיכרון, וכדומה.

- **הגנה ברמת המעבד**

יש להבטיח שמערכת ההפעלה תהיה מסוגלת להחזיר אליה את השליטה במעבד. כלומר לא יוצר מצב בו תהליך משתמש שולט ביחידת העיבוד ללא הגבלת זמן.

- למטרה זו משתמשים לרוב ב-timer.

- ניתן לכוון טיימר לכך שיצור פסיקה לאחר פרק זמן קצוב מראש.

- טיימר ממומש על ידי שעון ומונה (counter). בתחילת התהליך המונה מקבל מספר שלם חיובי, ומיד לאחר מכן מתבצעת ספירה לאחור (ברמת החומרה). כאשר ערך המונה 0 נוצרת פסיקה (על ידי ה-Timer בעצמו). השליטה עוברת למערכת ההפעלה והיא תבחר לפעול בהתאם.

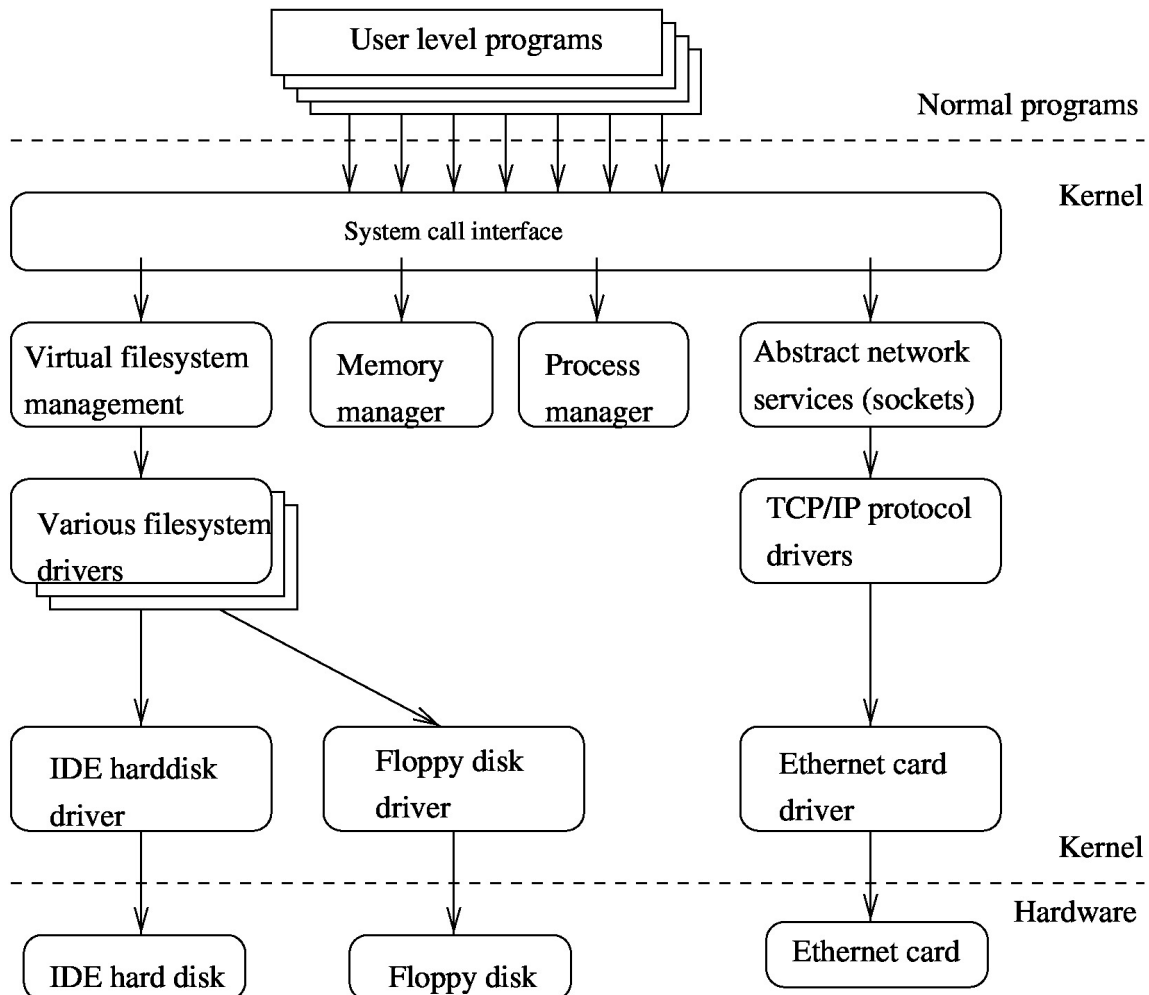
- בדרך כלל תהליך מוקצב זמן פעולה שלאחריו התהליך יעצר לחלוטין או יושהה ומערכת ההפעלה תעבור לתהליך הבא בתור.

- בצורה האחרונה ממומשת מערכת שיתוף זמן.

4.7 קריאות שרות (system calls)

- מאחר ופעולות קלט ופלט יכולים להתבצע רק על יד מערכת ההפעלה, נשאלת השאלה איך משתמש רגיל יוכל לבצע אותם?
- הפיתרון הוא שכל בקשה לבצע פעולות I/O על ידי תהליכי משתמשים צריכה להיות מופנית בצורה מסודרת למערכת ההפעלה. מערכת ההפעלה תמלא את הבקשה אם לתהליך המבקש אותה יש את ההרשאות המתאימות. מילוי הבקשה יתבצע על פי הכללים של מערכת ההפעלה.
- הכללים צריכים לפתור בעיות של תורים, תיזמון, ניצול יעיל של יחידת העיבוד, ועקביות עם בקשות אחרות.
- מנקודת המבט של התהליך הבודד, שיטת עבודה זו עשויה להיראות מסורבלת או לא הוגנת. למשל, אם כמה תהליכים מבקשים לכתוב לכמה מקומות שונים בדיסק, מערכת ההפעלה רשאית להשהות את הביצוע או ולבצע את רשימת הבקשות בכל סדר הנראה לה יעיל עבור פעולה תקינה או יעילה יותר של מערכת המחשב.
- כל בקשה כזו מצד התהליכים נקראת `system call` או `operating system function call`.
- בגירסת Linux של מערכת ההפעלה Unix קיימות למעלה מ-140 `system calls`. דוגמאות: `fork`, `link`, `close`, `open`, `write`, `read`.

● סיכום מבנה מערכת הפעלה



איור 4.5: מבנה סכימטי של מערכת הפעלה

פרק 5

מערכת ההפעלה Unix

5.1 היסטוריה

- נכתבה לראשונה בשפת סף על ידי Ken Thompson, Bell Labs, עבור מחשב PDP-7, כניסיון לפשט את מערכת ההפעלה Multics שאיכזבה.
- יותר מאוחר הצטרפו לפרוייקט Dennis Ritchie, Brian Kernighan, ועוד מספר חוקרים מאותה מחלקה. המערכת נכתבה מחדש למחשב PDP-11 שהיה יותר מתקדם.
- בשלב השני של ההתפתחות נקבעה ההחלטה בדבר השפה שבה תכתב המערכת. היה ברור כי כתיבה מחדש את כל מערכת עבור כל מחשב חדש לא באה בחשבון לטווח הארוך.
- תומפסון החליט לכתוב את רוב המערכת בשפה עילית שנקראה שפת B (שנגזרה משפה קודמת בשם BCPL).
- בגלל מגבלות שונות של שפת B ניסיון זה נכשל.

- דניס ריצ'י החליט לכתוב שפת תיכנות חדשה שתרחיב את שפת B אך לא תסבול מהמגבלות של שפת B - הוא קרא לשפה החדשה שפת C.
 - בנוסף לכך, דניס ריצ'י כתב קומפיילר מצוין לשפת C.
 - שפת C עמדה בכל הציפיות בהצלחה. תומפסון וריצ'י כתבו את Unix מחדש בשפת C.
 - החל משלב זה, העתקת Unix למחשבים חדשים הפך לעניין פשוט יחסית.
- פיתוח שפת C היה הדבר הנכון בזמן הנכון, ומאז כמעט כל מערכת הפעלה מודרנית נכתבת בשפה זו.
 - בתוך פרק זמן קצר של מספר שנים הפכה Unix למערכת ההפעלה הכי פופולרית באוניברסיטאות, מכללות, מכוני מחקר, מוסדות ממשלה, וחברות גדולות.
 - זוהי עדיין מערכת ההפעלה שרצה על המגוון הכי גדול של מחשבים. היא תוכננה בקפידה מוחלטת עם מטרות ברורות מאוד, ולמרות גילה היא עדיין מודרנית ואלגנטית.
 - אך הסיבה העיקרית ללימוד Unix במוסדות אקדמיים היא שרוב העקרונות של תורת מערכות ההפעלה מודגמים במערכת זו בצורה הטובה ביותר. בנוסף לכך, כל מערכות ההפעלה המצליחות שפותחו לאחר מכן (כגון Dos או WinNT) הושפעו

בצורה מכרעת על ידי Unix, ולכן לא ניתן להבין אותם ללא הבנה בסיסית של Unix.

- בניגוד למערכות הפעלה אחרות, Unix מותקנת תמיד עם כל תוכניות הקוד שלה (בשפת C). מבחינה זו היא אידיאלית עבור מחקר והוראה, ובכלל עבור כל מי שמעוניין לבצע שינויים מעניינים בגרעין המערכת.

- בשנת 1984, לאחר הפיצול של חברת AT&T על ידי ממשלת ארה"ב, החברה הפיצה את הגירסה המסחרית הראשונה של Unix: System III. המוצר לא התקבל טוב, ותוך שנה הוחלף במוצר משופר יותר: System V. מאוחר יותר גירסה זו עודכנה וגדלה בצורה ניכרת - System V Release 4

- אוניברסיטת ברקלי שאימצה את מערכת ההפעלה Unix התנגדה לכוונות המסחריות של AT&T ולקחה על עצמה פיתוח גירסה לא מסחרית של Unix. היא נעזרה לשם כך בתקציבים של סוכנות DARPA של משרד ההגנה האמריקאי

DARPA - Defence Advanced Research Projects Agency

אשר מאוחר יותר סייעה גם להקמת רשת האינטרנט.

- בניגוד לגירסה המסחרית של System V, הגירסה האחרונה של Berkeley, 4.4BSD (Berkeley Software Distribution), כללה מספר גדול של שיפורים והרחבות:

- זיכרון וירטואלי באמצעות paging.
 - שמות קבצים ארוכים (במקום 14 ב-System V).
 - מערכת קבצים משופרת ויותר מהירה.
 - מספר יותר גדול של תוכניות יישומים כגון העורך vi, מעבד הפקודות csh, קומפיילרים עבור Pascal ו-Lisp.
 - פרוטוקולי תקשורת TCP/IP שכיום הם הפרוטוקולים הסטנדרטיים של רשת האינטרנט.
 - מערכת סיגנלים אמינה יותר.
- כתוצאה מכך הפכה גירסת ברקלי ליותר פופולרית במוסדות אקדמיים וממשלתיים שהיו זקוקים לשרותי התקשורת הטובים שהיא ספקה.
 - חברות מסחריות כמו SUN Microsystems ו-DEC ביססו את גירסאות ה Unix שלהן על זו של Berkely. חברת SUN הפיצה את מערכת ההפעלה Solaris בתחנות העבודה שלה שנעשו יותר ויותר נפוצות מהמחשבים הגדולים הישנים.
 - ריבוי הגירסאות השונות של Unix יצר מבוכה מסוימת וגרם נזק להצלחה מסחרית שלה. שום ספק תוכנה לא יכל להבטיח כי מוצריו יפעלו בכל הסביבות הקיימות של Unix.
 - נעשו מספר נסיונות לתת הגדרה סטנדרטית אך הם נכשלו.

- הנסיון הרציני הראשון לפשר בין שתי הגירסאות הראשיות נעשה על ידי המכון IEEE:

IEEE - Institute for Electrical and Electronic Engineers

במסגרת מכון חשוב מאוד וניוטרלי זה התכנסו כמה מאות אנשים מהתעשייה, אקדמיה, ומוסדות ממשלתיים ועשו מאמץ גדול לתת הגדרות מדויקות לתנאים ההכרחיים שעל מערכת ההפעלה לקיים. התוצאה היא תקן בשם POSIX (Portable Operating System).

- התקן כלל הגדרות של פונקציות ספרייה שחייבות להיות חלק מכל גירסה של Unix. מרבית פונקציות אלה משתמשות בשגרות שרות של הגרעין, אך הן עשויות להיות מיושמות גם מחוץ לגרעין של המערכת. החלקים הטובים של BSD ושל SystemV נכנסו לתקן.

- הרעיון של POSIX הוא שכל ספק תוכנה המבסס את מוצריו על POSIX יוכל להבטיח שהן יפעלו בכל הגירסאות השונות של Unix.

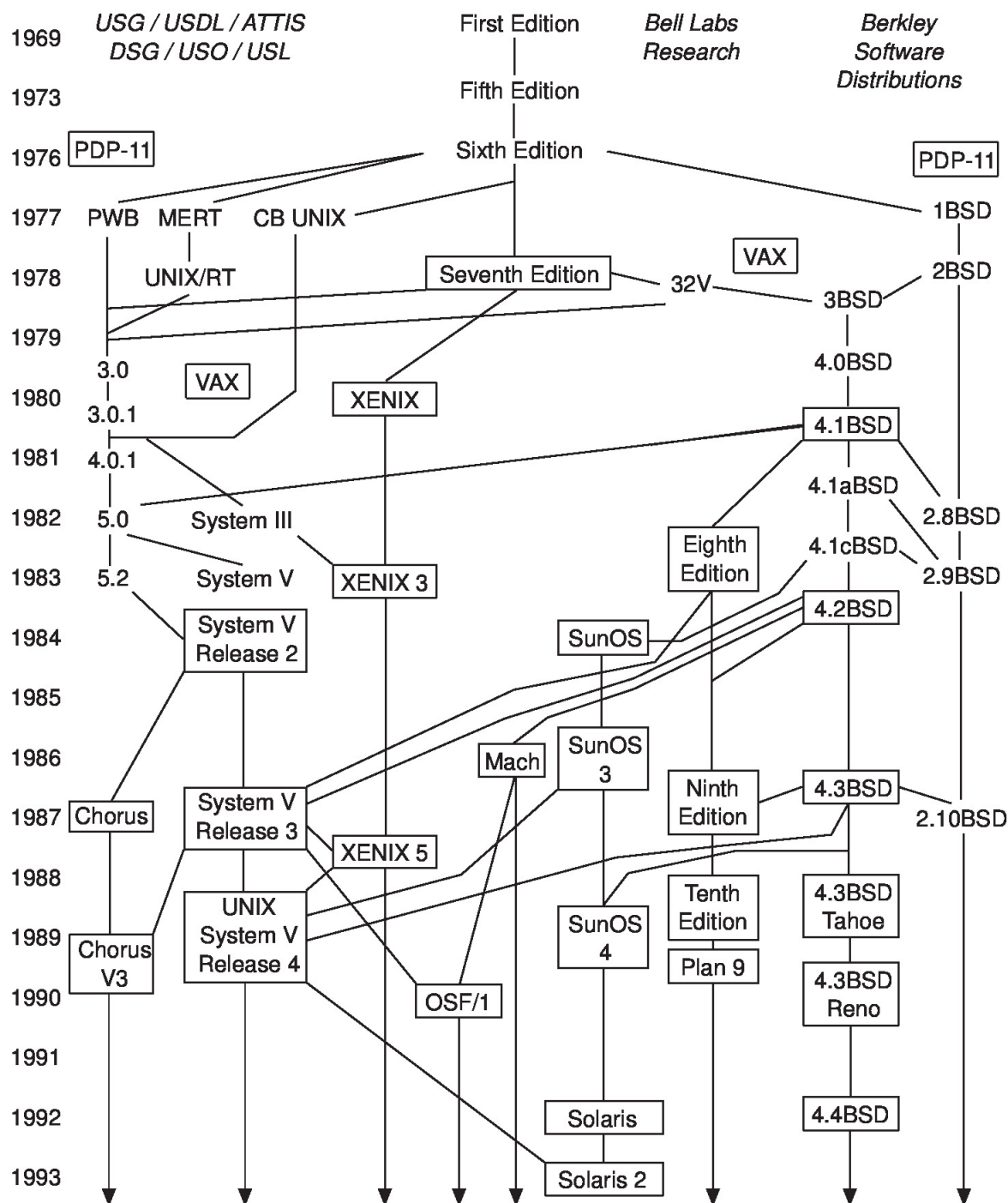
- נסיון זה הצליח במידה רבה. הגירסה האחרונה של SystemV, SVR4, כוללת בתוכה כמעט את כל השיפורים שנעשו בגירסת ברקלי.

- לרוע המזל, חברות כמו IBM, DEC, ו-HP לא אהבו את הרעיון שלחברת AT&T תהיה שליטה מלאה על Unix, והם הקימו

אגודה בשם OSF (Open Software Foundation), שאומנם אימצה את כל הסטנדרטים של POSIX, אך הוסיפה עליהם סטנדרטים נוספים משלה כגון: סטנדרטים למערכת חלונות (X11), מימשק גרפי למשתמש (Motif), תיכנות מבוזר, ועוד.

- התוצאה הכללית היא שכיום קיימים שני גופים תעשייתיים גדולים שמכתיבים סטנדרטים שונים של Unix. יותר מאוחר, חברת IBM הוציאה את גירסת AIX.
- גירסאות אלה גדלו למימדי ענק, כך שגם אם קוד המערכת פתוח לכולם, אין שום סיכוי שאדם אחד יוכל ללמוד ולהבין אותו בשלמותו. זה בניגוד גמור לעקרון היסודי של Unix: מערכת קטנה חזקה וניתנת להבנה וללימוד.
- מצב זה הוביל לנסיון ליצור מחדש גירסאות קטנות כמו Minix אשר קל ללמוד אותן, וכל הקוד פתוח לכל, במיוחד למטרות מחקר והוראה.

● היסטוריה של גרסאות Unix



איור 5.1: היסטוריה של מערכת ההפעלה Unix

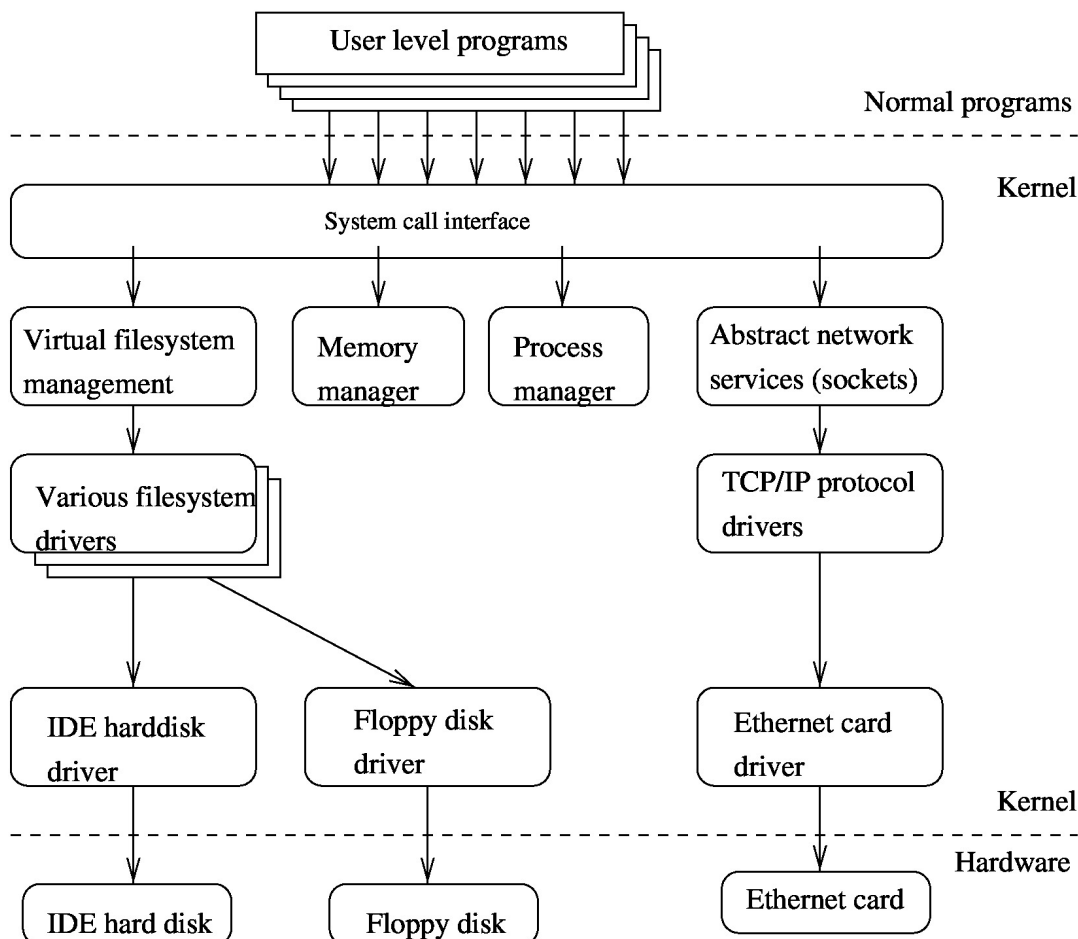
- במשך כמה שנים, גרסת Minix שפעלה גם על מחשבים אישיים מילאה תפקיד זה. בשנת 1991, המערכת Minix הורחבה למערכת ההפעלה Linux, וכיום Linux ממלאת תפקיד זה יותר

ויותר במוסדות האקדמיים.

- הגרעין הראשון פותח בשנת 1991 על ידי Linus Torvalds
- תוכננה לריצה מהירה על מחשבי PC
- כיום גם עבור Sparc, Alpha, PowerPc.
- אינה שייכת לשום גוף מסחרי
- ממשיכה להתפתח על ידי אלפי מתכנתים באמצעות רשת האינטרנט
- תומכת בפרוטוקול התקשורת TCP/IP
- כיום כעשרה ספקים שונים. הכי פופולריים: RedHat, SuSe, Ubuntu, Slackware, Caldera.
- multitasking, multiuser, ערכה מלאה של כלי Unix.
- מערכת ההפעלה Linux מבוססת בעיקר על SystemV, וכמובן מצייתת לכל הדרישות של POSIX. למרות זאת היא כוללת בתוכה את כל הדברים הטובים של BSD.

5.2 קריאות שירות (System Calls)

• מבנה סכימטי של מערכת ההפעלה Unix



איור 5.2: מבנה סכימטי של מערכת ההפעלה Unix

- בגירסת Linux של מערכת ההפעלה Unix קיימות למעלה מ-140 system calls. להלן מספר דוגמאות נפוצות:

pid = fork()	Create a child process
s = waitpid(pid, &status, opts)	Wait for a child to terminate
s = execve(name, argv, envp)	Replace a process' core image
exit(status)	Terminate a process and return status
fd = open(file,how)	Open a file for reading and/or writing
s = close(fd)	Close an open file
n = read(fd, buffer, nbytes)	Read data from file to buffer
n = write(fd, buffer, nbytes)	Write data from buffer to file
pos = lseek(fd, offset, whence)	Move the file pointer
s = mkdir(name, mode)	Create a new directory
s = rmdir(name, mode)	Delete an empty directory
s = link(name1, name2)	Create a new directory entry for file
s = unlink(name)	Remove a directory entry
s = chdir(dirname)	Change the working dir of a process
s = chmod(name, mode)	Change a file's protection bits

The return code s is 0 for success, -1 for error.

- יש להבחין בין system call ו-library function.
- למשל open היא פונקציית מערכת ואילו fopen היא פונקציית ספרייה המוגדרת באמצעות open ודברים נוספים.
- הפונקציות fopen, fclose, fgetc, fputc, fgets, fputs, וכו', הן פונקציות השייכות לספרייה stdio.h. ספרייה זו קיימת במערכות הפעלה שונות. במערכת ההפעלה Unix פונקציות אלה ממומשות על ידי פונקציות המערכת היסודיות של הגרעין של Unix.

- רוב הפקודות הסטנדרטיות של Unix הם למעשה תוכניות בשפת C הנעזרות בספריות המערכת.
- דוגמא: גירסה פשוטה מאוד של תוכנית C של הפקודה cp של Unix:

```
#include <stdio.h>
#include <fcntl.h>
#include <syscalls.h>
#define PERMS 0666 /* rwx for owner, group, others */
void error(char *, ...) ;
/* cp: copy f1 to f2 */
main(int argc, char $argv[])
{
    int f1, f2, n;
    char buf[BUFSIZE];
    if (argc != 3)
        error("Usage: cp file1 file2");
    if ((f1 = open(argv[1], O_RDONLY, 0)) == -1)
        error("cp: can't open %s", argv[1]);
    if ((f2 = creat(argv[2], PERMS)) == -1)
        error("cp: can't create %s, mode %03o", argv[2], PERMS);
    while ((n = read(f1, buf, BUFSIZE)) > 0)
        if (write(f2, buf, n) != n)
            error("cp: write error on file %s", argv[2]) ;
    return 0;
}
```

- אם נקמפל את התוכנית הזו נקבל גירסה פשוטה (ולא בטוחה) של הפקודה cp. התוכנית האמיתית של cp כוללת מעל 1200 שורות.

פרק 6

Unix - מושגי יסוד

6.1 פקודות שימושיות וסימנים מיוחדים

- רשימת פקודות בסיסיות לעבודה ראשונית במערכת Unix:

```
cd          - change directory
cp          - copy
mv          - move
mkdir       - make directory
rm          - remove file/dir
rmdir       - remove an empty directory
rm -r       - remove recursively
rm -i       - interactiv remove
ls -aCF     - list all files in columns
pwd         - print working directory
cat         - display file
more        - display file by pages
less        - display file by pages (page-up)
grep        - get regular expression
man         - manual pages
```

```
command > file
command < file
Special Characters: * ? " \ | . .. &
```

● פירוט:

```
*   matches any string
?   matches any character
"   quotation
\   cancels special characters
.   the current directory (soft link)
..  the parent directory (soft link)
&   run command in the background
```

● דוגמאות

```
cp /etc/passwd "The password file"
cp /etc/passwd \*.\?
ls -l /etc | more
ls -s /usr/bin | less
ls -s /usr/bin | sort | less
ls -l > file &
ls -l /tmp | grep dany
cat /etc/passwd | grep u3203903
grep u3203903 /etc/passwd
man ls
man man
cat file1 file2 file3 > file4 &
```

6.2 קבלת עזרה

- דפי עזרה של Unix מתחלקים לשמונה פרקים:

1. פקודות משתמש ברמת מעבד הפקודות
2. תוכניות מערכת (system calls) מתוך ספריות C
3. פונקציות נוספות מתוך ספריות C
4. התקנים ודרייברים "special files"
5. תאור סוגי קבצים
6. משחקים
7. שונות
8. תוכניות לתחזוק המערכת ומידע

- בדרך כלל ממוקמים בספריות:

```
/usr/man/man1  
/usr/man/man2  
/usr/man/man3  
....  
/usr/man/man8
```

- ברירת המחדל של הפקודה man היא פרק 1.
- חיפוש על פי מילת מפתח: `man -k keyword`

● דוגמאות

```
man 1 mkdir
man mkdir
man 2 mkdir
=> There is a system call mkdir!
man grep
man 5 passwd
=> information on the file /etc/passwd
man passwd
=> information about the command passwd
man -k password
=> All commands related to password
```

● אינפורמציה נוספת ניתן למצוא בספריות כגון:

```
/usr/doc
/usr/share/doc
/usr/share/info
/usr/info
/usr/doc/HOWTO
/usr/doc/HTML
/usr/doc/FAQ
```

- בסביבת החלונות X windows קיימת הפקודה xman לשם הצגת דפי העזרה בצורה גרפית ויותר נוחה לדיפדוף. בסביבה זו ניתן גם להשתמש בדפדפן אינטרנט בכדי לדפדף בדפי העזרה. יש להתחיל במסמך `./usr/doc/HTML/index.html`.

6.3 תהליכים

- ניתן לתאר את כל הפעילות של המערכת במונחים של תהליכים.
- קיימים תהליכי המערכת: הגרעין של המערכת, ומספר תהליכים קבועים לטיפול בדואר תיקשורת, ופעולה שוטפת של המערכת.
- כל משתמש יכול להריץ מספר תהליכים בו-זמנית. כל תהליך שנוצר בדרך זו עשוי להוליד תהליכים נוספים (ילדים) במהלך פעולתו.
- כל תהליך שייך למשתמש מסוים. תהליכי מערכת ההפעלה שייכים ל-root.
- כל משתמש יכול להציג על המסך את רשימת התהליכים השייכים לו באמצעות הפקודה ps.

```
ps          - list my processes
ps a        - list processes of all users
ps au       - list all processes in user format
ps auw      - as above in wide format
```

- דוגמא (מקוצרת) לפלט של הפקודה ps auw:

USER	PID	%CPU	%MEM	SIZE	RSS	TTY	STAT	START	TIME	COMMAND
root	222	0.0	2.5	1440	776	1	S	18:44	0:00	/bin/login
root	230	0.0	3.0	1316	928	1	S	18:53	0:00	-zsh
root	240	0.0	1.9	1120	612	1	S	18:53	0:00	startx
root	319	0.0	3.8	2048	1180	1	S	18:53	0:01	control-panel
root	329	0.0	5.2	2408	1620	1	S	18:54	0:00	xterm
root	330	0.0	3.1	1320	972	p1	S	18:54	0:00	zsh

```

root  1329  0.0  1.5   820   488  p1 R   23:34  0:00  ps auw
dany  446   0.0  2.5  1444   784  p4 S   19:15  0:00  login
dany  447   0.0  2.4  1160   772  p4 S   19:15  0:00  -bash
dany  1211  0.0  2.5  1168   776  p5 S   22:41  0:00  vi file.txt
dany  1370  0.8 30.6 10160  9488  p5 T   00:25  0:00  gnuchessx
miki  1398  0.0  2.5  1428   780  p8 S   00:31  0:00  /bin/login
miki  1399  0.1  2.5  1184   792  p8 S   00:31  0:00  -bash
miki  1437  1.1  4.7  2312  1480  p8 S   00:40  0:00  xcalc
miki  1440 11.8 28.8 10096  8920  p8 S   00:42  0:00  gnuchessx

```

● הסברים:

PID - process identity number
 CPU - central proceessing unit time
 SIZE - size of text+data+stack
 RSS - kilobytes of program in memory
 TTY - controling terminal (teletype)
 STAT - status of the process:
 S sleeping
 T stopped
 R running (or ready to run)
 Z zombie

● **תרגיל:** כנס למערכת, הפעל את הפקודה "ps auxw", וערוך רשימה מלאה של כל התהליכים השייכים ל-root. על ידי שימוש בפקודה man נסה להבין בערך מהם התפקידים של כל התהליכים האלה.

● **סיום תהליכים באמצעות הפקודה kill**
 כל משתמש יכול לסיים כל תהליך השייך לו בלבד.

```
~/games> ps
PID TTY STAT  TIME COMMAND
398  p5 S    0:00 /bin/login -h mail.braude.ac.il
399  p5 S    0:00 -bash
418  p6 S    0:00 bash
420  p5 S    0:00 xboard
421  p5 R    4:12 gnuchessx 40 5
436  p5 R    0:00 ps

~/games> kill 420

~/games> ps
PID TTY STAT  TIME COMMAND
398  p5 S    0:00 /bin/login -h localhost -p
399  p5 S    0:00 -bash
418  p6 S    0:00 bash
444  p5 R    0:00 ps
```

- שאלה: למה הסתיים תהליך 421 למרות שבקשנו רק את סיום תהליך 420?
- הערה: במידה ותהליך מסוים נתקע ואינו משחרר את המסך, יש צורך להתחבר לחשבון דרך מסך אחר ומשם לסיים את התהליך התקוע.
- **סיגנלים (signals)**
 בכדי לשלוט על תהליכים, ניתן לשלוח אליהם סיגנלים בכדי ליידע אותם לגבי התרחשות אירועים מיוחדים או לבקש אותם לסיים. למעשה זוהי המטרה העיקרית של הפקודה `kill`.

● דוגמא:

```

~> ps
  PID TTY STAT  TIME COMMAND
  398 p5 S    0:00 /bin/login -h localhost -p
  399 p5 S    0:00 -bash
  450 p6 S    0:00 bash
  497 p5 S    0:00 vi targil.text
  499 p6 S    0:00 xboard
  500 p6 S    0:00 gnuchessx 40 5
  502 p6 R    0:00 ps
~> kill -s SIGKILL 497
better to type
~> kill -s 9 497

```

● טבלת סיגנלים נפוצים

1	SIGHUP	Terminal hangup
2	SIGINT	Terminal interrupt
3	SIGQUIT	Terminal quit
9	SIGKILL	Process killed
13	SIGPIPE	Broken pipe
14	SIGALARM	Alarm clock interrupt
15	SIGTERM	Software termination
23	SIGCONT	continue job if stopped
25	SIGSTP	interactive stop signal

● כל תהליך יגיב לסיגנלים הנשלחים אליו בהתאם לקוד התוכנית שלו (signal handlers). מלבד ממקרים מיוחדים, תהליך עשוי "לתפוס" את הסיגנל הנשלח אליו ולהתעלם ממנו. שום תהליך אינו יכול להתעלם מהסיגנל SIGKILL (וגם SIGSTP). לכן

סיגנל זה יכול לסיים כל תהליך המסרב להסתיים בצורה נורמלית (SIGTERM).

- קיימים בערך 30 סיגנלים שונים.
- כל מעבד פקודות מודרני מספק למשתמש כלים לשם "בקרת תהליכים" (job control). במעבד הפקודות bash למשל ניתן לבצע את הדברים הבאים:

```
Control-Z  stop a job, keep it in the
            background, and return to
            the shell
bg         continue the last stopped job
            in the background
bg %n     continue the n-th stopped job
            in the background
fg        return to the last stopped job
fg %n     return to the n-th stopped job
jobs      list of all jobs that are running
            in the background
```

- **תרגיל:** השתמש בעורך הטקסט vi בכדי לערוך 5 קבצים פשוטים בו-זמנית. על ידי שימוש בפקודות הנ"ל עבור מעריכה הקובץ החמישי לקובץ השני.

- **משתנים סביבתיים (environment variables)**

- לכל תהליך יש קבוצה של משתנים סביבתיים.

- תהליכים מורשיים את המשתנים הסביבתיים שלהם להתהליכים שהם יוצרים.

- הפקודה env

מעבד הפקודות bash, למשל, הוא התהליך הראשון שמופעל עבור המשתמש כאשר הוא נכנס למערכת והוא בעל קבוצה חשובה של משתנים סביבתיים.

- ניתן לקבל את רשימת המשתנים הסביבתיים הסטנדרטיים של מעבד פקודות זה על ידי הפקודה env או printenv.

```

~> env
ENV=/home/cs/stud97/u3203903/.bashrc
HISTSIZE=1000
HOSTNAME=mail.braude.ac.il
LOGNAME=u3203903
HISTFILESIZE=1000
MAIL=/var/spool/mail/u3203903
TERM=xterm
HOSTTYPE=i686
PATH=/usr/local/bin:/bin:/usr/bin:./:/usr/X11R6/bin
HOME=/home/cs/stud97/u3203903
SHELL=/bin/bash
PS1=\w>
USER=u3203903
DISPLAY=mail.braude.ac.il:0.0
OSTYPE=Linux
MM_CHARSET=ISO-8859-1
SHLVL=1

```

• משתנים אלה עוברים בירושה לכל תהליך שנוצר באמצעות מעבד הפקודות.

- ניתן לקבל את התוכן של משתנה סביבתי על ידי הוספת התו \$ לפניו. מעבד הפקודות יבצע קודם את החלפת המשתנה בתוכנו, ורק לאחר מכן יפעיל את הפקודה שתקבל כתוצאה מכך.

```
~> more $MAIL - read your mail
```

```
~> echo $PATH - see your path
```

```
~> PATH=$PATH:$HOME/bin
```

```
~> cp $MAIL mymail
```

```
~> grep -i "Yoram Cohen" $MAIL
```

6.4 מערכת הקבצים

- לקובץ במערכת Unix אין שום מבנה מיוחד: קובץ הוא סדרת בתים (bytes).
- כל קובץ מיוצג במערכת על ידי מספר: index number. מנקודת המבט של מערכת ההפעלה, המספר הזה הוא הנציג האמיתי של הקובץ.
- ספרייה היא קובץ מיוחד המכיל טבלה המקשרת בין שמות רגילים של קבצים ובין מספריהם. השמות הרגילים מיועדים עבור במשתמשים, אך מערכת ההפעלה עצמה עובדת עם המספרים שלהם.

Directory Contents

.	5826
..	7610
raedme.txt	13093
targil8	22703
targil5	8391
mailbox	6326
dany.txt	9273
dir2	12091
directory8	2174

איור 6.1: רשומה של תיקיית Unix

- לאותו הקובץ עשויים להיות כמה שמות שונים. למשל שני משתמשים שונים יכולים להיות שותפים לאותו הקובץ:

```
user1> edit /home/ee/user1/contacts/phones.txt
user2> ln ~user1/contacts/phones.txt user1_phones.txt
=> Now user2 has a link to ~user1/contacts/phones.txt
This file has one index number, but two names!
(2 links)
```

- ישנם כמה סוגי קבצים:

א. קבצים רגילים - regular files

ב. ספריות - directories

ג. קבצים מיוחדים: device special files, fifo special files,

sockets

- שם קובץ יכול להיות מורכב מלכל היותר 255 תווים. התו / אינו יכול להופיע בשם קובץ. תוים נוספים עשויים להיות בעייתיים מאחר שמעבד הפקודות מפרש אותם בצורה מיוחדת. קיימת הבחנה בין אותיות אנגליות גדולות לקטנות.
- נקודה בתחילת שם קובץ הופכת אותו לקובץ נסתר. בכדי להציג את רשימת כל הקבצים, כולל הקבצים הנסתרים, יש להשתמש בפקודה `ls -a`.
- **תרגיל:** בדוק מה הם הקבצים הנסתרים בספריית הבית שלך ומהו תוכנם?
- מסלולים אבסולוטיים ומסלולים יחסיים: לכל תהליך במערכת Unix יש ספריית עבודה נוכחית אשר בה הוא מתבצע. ניתן לציין מסלולי קבצים יחסית לספרייה זו או באופן אבסולוטי.
- עץ הקבצים של Unix עשוי להיות מורכב מכמה מערכות קבצים שונות השוכנות בהתקנים שונים - דיסקים קשיחים נפרדים, `cdrom`, מחיצות שונות של אותו דיסק קשיח, ואפילו דיסקטים.
- סידור כזה נוח למטרות אדמיניסטרציה. למשל, על מחיצה אחת ניתן להתקין את כל תוכניות המשתמשים, על מחיצה שניה את כל חשבונות המשתמשים, וכו'. מאוחר יותר ניתן לשדרג את המערכת או לגבות את חשבונות המשתמשים ביתר נוחות.
- מערכות הקבצים יכולות להיות שונות גם מבחינת המבנה שלהן.

למשל, ניתן לשלב מחיצות של Dos או Win95 בתוך עץ הקבצים של Unix.

- לא ניתן לבצע קישור (link) בין קבצים השייכים לשתי מערכות קבצים שונות.
- בכדי לפתור בעייה זו פותח רעיון הקישור הסימבולי (symbolic link) קישור כזה נקרא לפעמים גם "soft link", וקישור רגיל נקרא "hard link".
- קישור סימבולי מתבצע על ידי הפקודה `ln -s`.
- קישור סימבולי הוא קובץ אשר תוכנו הוא מסלול של קובץ או ספרייה. הקובץ המקושר עשוי להשתייך למערכת קבצים שונה. ניתן לזהות קישורים סימבוליים על ידי הפקודה `ls -l`.
- הנתונים של הקובץ נשמרים בדיסק כסדרה של בלוקים (מספר שלם של יחידות דיסק). גודל טיפוסי של בלוק הוא 4096 בתים. סדרת הבלוקים אינה חייבת להיות רציפה על הדיסק, ובדרך כלל קבצים עשויים להתפזר בצורה אקראית על הדיסק.

```

~> ln -s /usr/doc/HTML info
~> ln /var/spool/mail/u3203903 MyMail
~> ls -l
total 16
drwx--x--x  2 user32 602   1024 Mar 30 18:40 Exams
-rw-rw----  2 user32 mail   2497 Mar 30 18:54 MyMail
-rw-r--r--  1 user32 602    247 Mar 30 16:48 file1
-rw-r--r--  1 user32 602    591 Mar 30 16:49 file2
drwxr-xr-x  2 user32 602   1024 Mar 31 00:47 games

```

```
lrwxrwxrwx 1 user32 602 13 Mar 31 11:07 info -> /usr/doc/HTML
~> cd info
~/info> netscape index.html
```

● כל קובץ במערכת הקבצים מיוצג על ידי מבנה נתונים שנקרא `inode`. בתוך מבנה זה מערכת ההפעלה שומרת את כל הפרטים הנחוצים לה לגבי הקובץ:

- גודל הקובץ.

- בעל הקובץ (uid)

- הקבוצה אליה משתייך בעל הקובץ (gid)

- זמן השינוי האחרון: mtime

- זמן הגישה האחרון: atime

- זמן השינוי האחרון של מאפייני הקובץ (בעלים, הרשאות, וכדומה): ctime

- הרשאות (?rwxrwxrwx, ?=-dlsbc)

- מספר הקישורים הקיימים לקובץ

- סוג הקובץ: קובץ רגיל, ספרייה, device file, fifo

- 12 מצביעים ל-12 הבלוקים הראשונים של הקובץ

- מצביע לבלוק של מצביעים לבלוקים נוספים של הקובץ

- מצביע לבלוק של מצביעים לבלוקים של מצביעים לבלוקים נוספים של הקובץ

triple indirect pointer -

size (in bytes)		
uid	gid	permissions
times: atime (access time) mtime (modification time) ctime (creation time)		
file type: regular file, directory, device file, fifo		
links		
12 direct pointers to blocks 1 single indirect pointer 1 double indirect pointer 1 triple indirect pointer		

איור 6.2: רשומה של קובץ במערכת ההפעלה Unix

- כל המבנים האלה מרוכזים בטבלת הקבצים הכללית של המערכת בה מתבצעת התאמה בין מספר הקובץ (index number) ובין מבנה הנתונים המתאים לו (inode).

inodes table

893	inode structure
1274	inode structure
2810	inode structure
7962	inode structure
14087	inode structure
35811	inode structure

איור 6.3: טבלת inodes במערכת ההפעלה Unix

● נפח מקסימלי של קובץ

חשבון פשוט מראה כי הגודל המקסימלי של קובץ ב-Unix הוא 4TB:

השיטה	מס בלוקים	גודל בלוק	הנפח המקסימלי
D	12	4K	48KB
SI	$1K \times 1$	4K	4MB
DI	$1K \times 1K \times 1$	4K	4GB
TI	$1K \times 1K \times 1K \times 1$	4K	4TB

● מחיצות (Buffers)

- מחיצה היא שטח בזיכרון המכיל נתונים שנקראו מקובץ או עומדים להכתב לתוך קובץ.
- כמעט כל תוכנית Unix משתמשת במחיצות בכדי לטפל בקבצים. למשל:
- התוכנית cp משתמשת במחיצה לשם העתקת קובץ.
- העורך vi משתמש במחיצה גדולה (עבור כל הקובץ!) לעריכת קובץ.
- מספר המחיצות וגודלם תלוי בתוכנית.

- הרשאות (Permissions)

Permissions

user			group			other		
r	w	x	r	w	x	r	w	x

איור 6.4: טבלת הרשאות במערכת ההפעלה Unix

- משמעות ההרשאות לגבי קובץ רגיל:

r מאפשר לקרוא ולהעתיק את הקובץ (לכל מי שיש לו הרשאה זו, user, group, או other).

w מאפשר לשנות את הקובץ (לכל מי שיש לו הרשאה זו, user, group, או other).

x מאפשר הפעלה של הקובץ במידה והקובץ ניתן להפעלה כתוכנית בינארית או תוכנית סקריפט.

- משמעות ההרשאות לגבי ספרייה:

r מאפשר להציג את רשימת הקבצים בספרייה. לא מאפשר להוסיף או למחוק קבצים בספרייה. לא מאפשר לקרוא או לשנות בקבצים קיימים.

w ניתן להוסיף או למחוק קבצים בספרייה. לא מספיק בכדי לאפשר לקרוא או לשנות בקבצים קיימים.

x מאפשר לבצע כל פעולה על הקבצים בספרייה, בתנאי ששמותיהם ידועים. מאפשר להיכנס לספרייה.

● הסיבית Set-uid

מאפשרת לתוכנית לרוץ עם ההרשאות של בעליה. דוגמא: התוכנית passwd שייכת ל-root, אך כל משתמש אחר רשאי להפעילה. תוכנית זו עשויה לבצע שינויים בקובץ /etc/passwd השייך גם הוא ל-root ואין הרשאת כתיבה למשתמשים אחרים. בכדי שמשתמש רגיל יוכל בכל זאת לשנות את סיסמתו, בזמן ההפעלה של הפקודה passwd הוא מקבל את ההרשאות של root.

```
/usr/bin/passwd    -r-sr-xr-x    root
/etc/passwd       -rw-r--r--    root
```

● הפעלת סיבית זו תבצע על ידי הפקודה:

```
chmod u+s program
```

היא תתפוס את המקום של x בשורת ההרשאות.

● באופן דומה קיימת הסיבית Set-Gid (במקום x של הקבוצה).

● הפקודה chmod

```
syntax:  chmod [-Rcfv] mode file
```

```
two types of modes: symbolic, numeric
```

```
-----
BEFORE
```

```
COMMAND
```

```
AFTER
-----
```

```
-----
```

```
chmod a=rw file
```

```
rw-rw-rw-
```

```

rw-----      chmod go+r file      rw-r--r--
rwxrwxrwx      chmod a-x  file      rw-rw-rw-
rwxrwxrwx      chmod g-w,o-wx file  rwxr-xr--
rwxrwx---      chmod o=g  file      rwxrwxrwx
rwxrwx---      chmod o=g-xw file     rwxrwxr--
rwxr-xr-x      chmod u+s  file      rwsr-xr-x
rwxrwxrwx      chmod 711  file      rwx--x--x
rwxrwxrwx      chmod 644  file      rw-r--r--
rwxrwxrwx      chmod 755  file      rwxr-xr-x

```

```

u  user
g  group
o  other
a  all  (equivalent to ugo)
+  add permission
-  deny permission
=  set exact permissions

```

● הפקודה umask

קבצים שנוצרים על ידי תוכניות שונות כגון vi או gcc, מקבלים מהמערכת שורת הרשאות rw-rw-rw- אם הם קבצים רגילים, או rwxrwxrwx אם הם קובצי הפעלה. ברירת מחדל זו ניתנת לשינוי על ידי הפקודה umask. בדרך כלל קובץ האיתחול של כל מעבד פקודות כולל את הפקודה umask 022. בצורה סימבולית:

```

umask -S u=rwx,g=r,o=r
vi file.txt
ls -l file.txt
-rw-r--r--  user51  603 ...

```


• תווים מיוחדים לתאור קבצים

* matches any string
? matches any single char
[...] matches all chars inside
[abg03] matches all chars a, b, g, 0, 3
[a-z] matches all chars from a to z
[^a-z] all chars different from a-z
[!a-z] all chars different from a-z
[a-z0-9] matches any alpha/numeric character

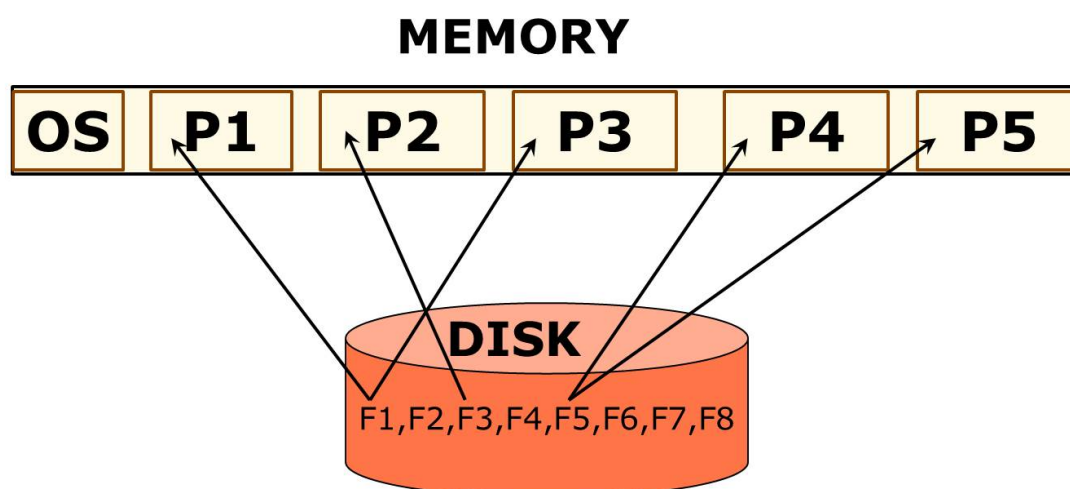
• ספריות חשובות

/usr /usr/bin /usr/local/bin /usr/bin/X11
/bin /sbin /usr/sbin /home /dev /lib
/usr/lib /usr/local/lib /usr/src /usr/include
/tmp /var /var/log /var/spool

פרק 7

זיכרון וירטואלי

- בין שאר תפקידיה הרבים והמגוונים גם מערכת הזיכרון במחשב מנוהלת על ידי מערכת ההפעלה.
- מערכת הזיכרון מתחלקת בין מערכת ההפעלה עצמה (שכיום תופסת חלק נכבד של הזיכרון הפיזי הראשי) ובין תהליכי המשתמשים השונים הרצים במערכת באותו הזמן (חלקם במצב של wait או מצב ready).



איור 7.1: מערכת דפים בזיכרון וירטואלי

7.1 משימות עקריות בניהול זיכרון

- על מערכת ההפעלה להקצות גודל מתאים של זיכרון לכל תהליך בתחילת ריצתו.
- מערכת ההפעלה חייבת למנוע מכל תהליך לפלוש למרחב הזיכרון של תהליך אחר, ובפרט למנוע מצב בו תהליך משתמש יוכל לפלוש למרחב הזיכרון של מערכת ההפעלה עצמה!
- על מערכת ההפעלה למנוע כל אפשרות שתהליך אחד יוכל לכתוב או לקרוא זיכרון של תהליך שני.
- מערכת ההפעלה תשחרר את הזיכרון של תהליך לאחר שסיים את פעולתו.
- **שיחלוף (Swapping):** טיפול במצב בו לא נותר בזיכרון מקום עבור תהליכים חדשים. על מערכת ההפעלה להעביר תהליכים בלתי פעילים (או ברמת תיעדוף נמוכה) מהזיכרון לדיסק ובכך לפנות מקום בזיכרון לתהליכים חדשים.
- זיכרון של תהליך שלא רץ במשך זמן ארוך יועבר למקום זמני בדיסק (swap-out).
- תהליך שהועבר לדיסק וצריך לחזור לרוץ, יועבר מהדיסק חזרה לזיכרון (swap-in).
- כל מערכת הפעלה מודרנית כוללת מנגנון **שיחלוף (swapping)**

קובץ שיחלוף (swap file) אשר בדרך כלל נמצא במקום ראשי בדיסק.

- במערכת ההפעלה Windows שני קבצים נסתרים `pagefile.sys`, `swapfile.sys`, בהם מאוחסנים דפי זיכרון בלתי פעילים לפרקי זמן קצרים או ארוכים (עד לטעינה מחדש לזיכרון במידת הצורך).

- קובץ השיחלוף (swap file) אינו קובץ רגיל. יש לו מבנה מיוחד מאוד המאפשר למערכת ההפעלה לבצע פעולות שיחלוף בקצב מהיר יותר מקצב כתיבה וקריאה בקבצים רגילים. אך עדיין, רחוק מאוד ממהירות כתיבה וקריאה בזיכרון הפיזי (באופן משמעותי מאוד).

- תהליך מתחיל בטעינה של קובץ הפעלה (executable file) מתוך הדיסק לזיכרון, ולאחר מכן הקצאת שטחי `Heap`, `Data` ו-`Stack`.

- כזכור, קובץ הפעלה יחיד (כגון `firefox.exe`) עשוי להיות מופעל מספר פעמים ובכך להפעיל מספר תהליכים שונים בזיכרון

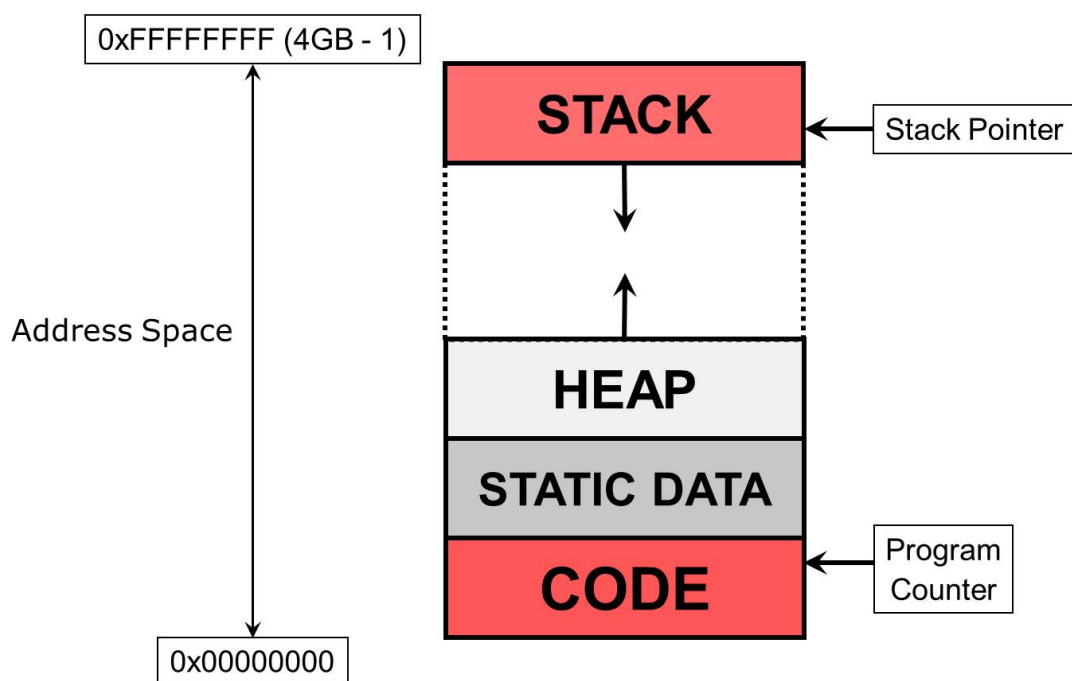
- בדיאגרמה שלמעלה רואים כי קובץ ההפעלה `F1` שימש לאיתחול שני התהליכים `P1`, `P3`.

- מהדיאגרמה רואים גם כי לכל תהליך הוקצה אזור נפרד בזיכרון, אך עדיין אין זה מבטיח כי תהליך אחד לא יוכל לפלוש

לשטחו של תהליך שני ללא השגחה קבועה וקפדנית של מערכת ההפעלה.

- למרות זאת, נציין כי קיימים מקרים חריגים בהם שני תהליכים שמתפיים פעולה (cooperating processes) ישתפו ביניהן שטח זיכרון משותף. אך זה מתאפשר רק על ידי הסכמה הדדית והקפדה על הליך מסודר של קריאות מערכת מתאימות.

7.2 מרחב כתובות וירטואלי

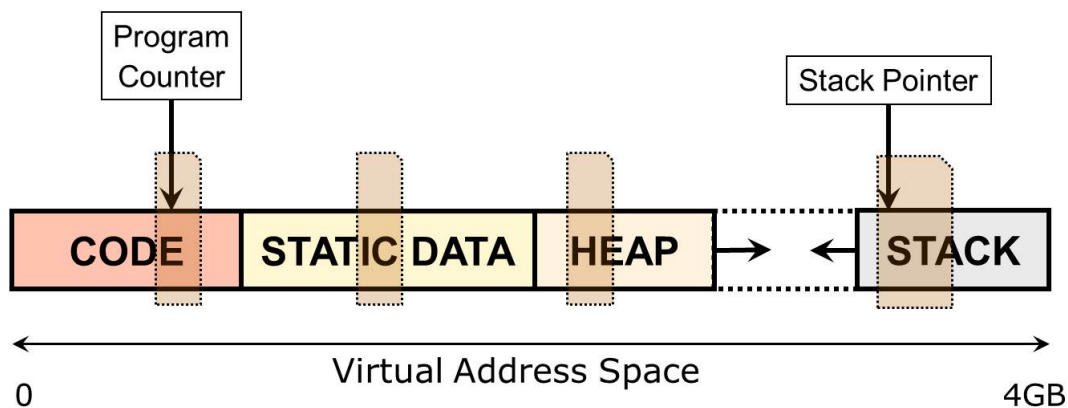


איור 7.2: מרחב כתובות בזיכרון וירטואלי

- **מרחב הכתובות של תהליך (Process Address Space)** הוא טווח הכתובות בהן התהליך עשוי בעיקרון להשתמש (באמצעות גודל התוכנית או על ידי הקצאה דינמית בזמן ריצה).

- לרוב מדובר בטווח המקסימלי שמתאפשר על ידי מצביע לכתובת, או ליתר דיוק, אוגר התוכנית (Program Counter Register) שגודלו השכיח הוא 32 סיביות. אוגר זה מסוגל לנוע מכתובת $0x00000000$ עד ל- $0xFFFFFFFF$ (4GB).
- ברור לכן כי מדובר במושג לוגי מופשט ולא בגודל פיזי אמיתי של זיכרון. מסיבה זו, מרחב הזיכרון הזה נקרא לפעמים גם **זיכרון וירטואלי (Virtual Memory)**.
- מחשב שכיח מכיל בדרך כלל זיכרון פיזי יחיד בגודל של 4GB, המיועד עבור כל התהליכים שרצים עליו. המחשב מסוגל להריץ מאות תהליכים במקביל, כשכל התהליכים נמצאים בו-זמנית בזיכרון הפיזי היחיד.
- ברור לכן שכמות הזיכרון המוקצית לכל תהליך כזה חייבת בהכרח להיות קטנה בהרבה מסדר גודל של 4GB.
- יתרה מזאת: במערכות של 64 סיביות, מרחב הכתובות של התהליך עשוי להגיע לגודל של 256TB, כאשר הזיכרון הפיזי של המחשב קטן מ-4GB.
- במקרה כזה ברור לחלוטין, שאין שום סיכוי להתאמה פשוטה בין המרחב הוירטואלי (או הלוגי כפי שהוא נקרא לפעמים) של התהליך ובין מרחב הזיכרון הפיזי בפועל (כפי שהוא באמת על הזיכרון הפיזי).

- בכדי להבין את מגמת הדיון, כדאי לשאול מספר שאלות בנוגע לטעינה של תוכנית לזיכרון:
 - א. האם יש צורך לטעון את כל התוכנית לזיכרון? בפרט אם היא מאוד גדולה?
 - ב. האם שטחי ה-Data, Heap, Stack חייבים להמצא בשלמותם בזיכרון?
 - ג. אם ניגשנו לקטע קוד או נתונים, האם נחזור אליו בשלב מאוחר יותר? ואם לא, האם יש טעם שיישאר בזיכרון?



איור 7.3: מרחב זיכרון וירטואלי של תהליך

- בחינה של שאלות אלה תוביל אותנו לאבחנות הבאות:
- בכל זמן נתון, התהליך משתמש רק בחלון קטן מאוד של התוכנית, הנתונים, והמחסנית.
- מרבית זיכרון התהליך אינו בשימוש במרבית הזמן.

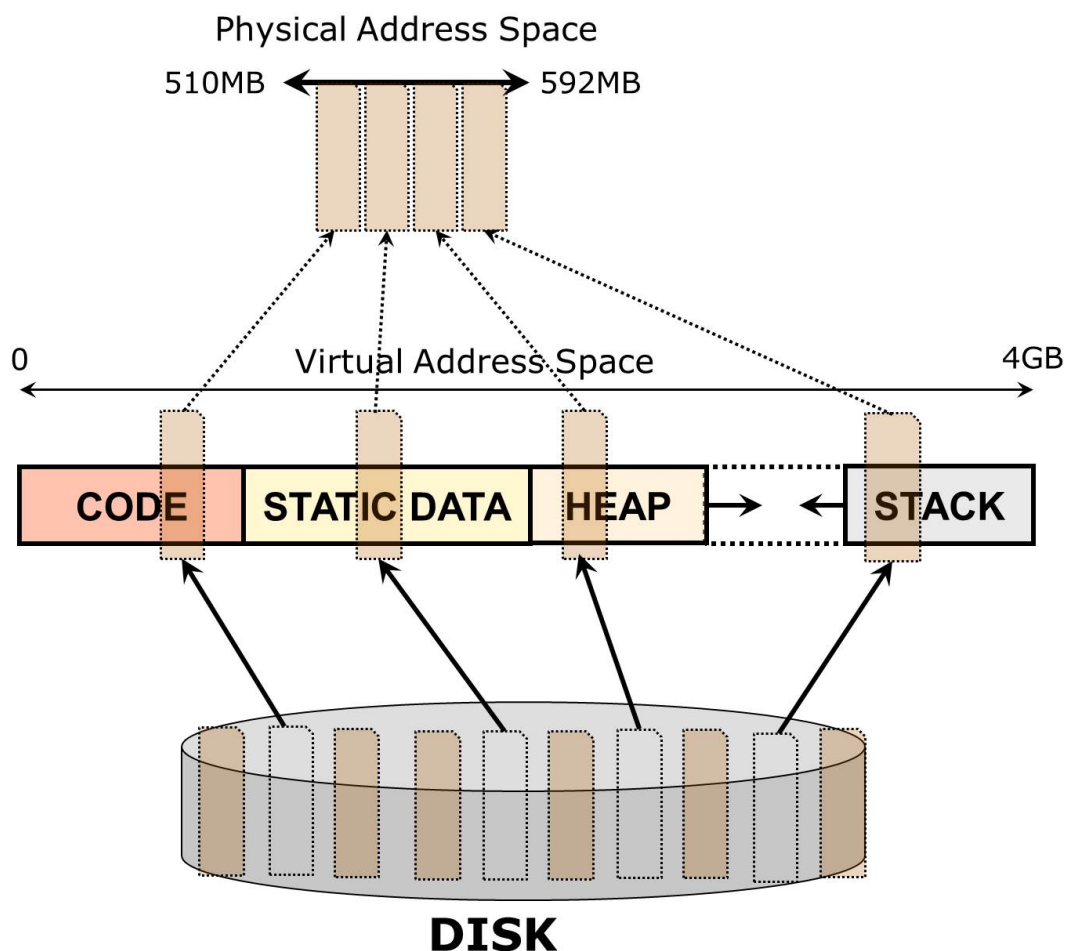
- מן הראוי שהקצאת זיכרון תתבצע רק למטרת שימוש מיידי (או לטווח הקצר) על ידי התהליך. נתונים שלא יעשה בהם שימוש בטווח המיידי, מן הראוי שיישמרו על הדיסק.
- הקצאת זיכרון עבור שימוש בעתיד הרחוק אינה מועילה ואינה רצויה
- קוד או נתונים בזיכרון, אשר לא נעשה בהם שימוש לאחרונה, רצוי להעביר מהזיכרון לדיסק. (לפחות באופן זמני עד שנזדקק להם שוב)

7.3 דיפדוף (Paging)

- השיטה המודרנית למימוש האבחנות האלה במערכת ההפעלה נקראת שיטת הדיפדוף (paging system).
 - מרחב הזיכרון הוירטואלי של המעבד מחולק לדפים (pages) בגודל בלוק של דיסק (בדרך כלל 4K), והזיכרון הפיזי מחולק למסגרות (Frames), כשגודלה של כל מסגרת שווה כמובן לגודלו של דף.
 - לכל תהליך יש מרחב כתובות מקסימלי המתאים לארכיטקטורה של המחשב.
- א. במערכת של 32 סיביות מרחב הכתובות נפרש על פני 4GB

ב. מערכות של 64 סיביות, מרחב כתובות מגיע עד 256TB

- אך רק הדפים הנחוצים לריצה של התהליך בטווח הקצר נמצאים בזיכרון הפיזי.
- למשל: במערכת של 32 סיביות, מרחב הכתובות יכול 1,048,576 דפים של 4K כל אחד. אך ייתכן בהחלט שעבור הריצה הנוכחית של התהליך דרושים לו רק חמשת הדפים הבאים: page8, page3, page41, page58, page93, כך שצריכת הזיכרון האמיתית של התהליך היא 20K ולא 4GB!



איור 7.4: מרחב כתובות פיזי מול וירטואלי

- בניגוד למרחב הכתובות הוירטואלי, מרחב הזיכרון הפיזי מוגבל על ידי הזיכרון הראשי וגודל הדיסק (שמשמש כהרחבת זיכרון לאחר שהזיכרון הראשי נוצל לחלוטין).
- לכל תהליך, מערכת ההפעלה מחזיקה טבלה שממפה בין מספר דף ומספר המסגרת (frame) המתאימה לו בזיכרון (Page Table).ble)
- בהינתן כתובת a יש למצוא את מספר הדף p שאליו הכתובת שייכת, ואת הקיזוז מתחילת הדף (offset). באמצעות טבלת הדפים נגלה את מספר המסגרת $f = \text{frame}(p)$ שמתאים לדף p . הפונקציה $\text{frame}()$ היא פונקציית התירגום מהדף הוירטואלי למסגרת בזיכרון הפיזי.
- הכתובת הפיזית תתקבל על ידי הנוסחה:

$$m = \text{frame}(p) * 4K + \text{offset}$$

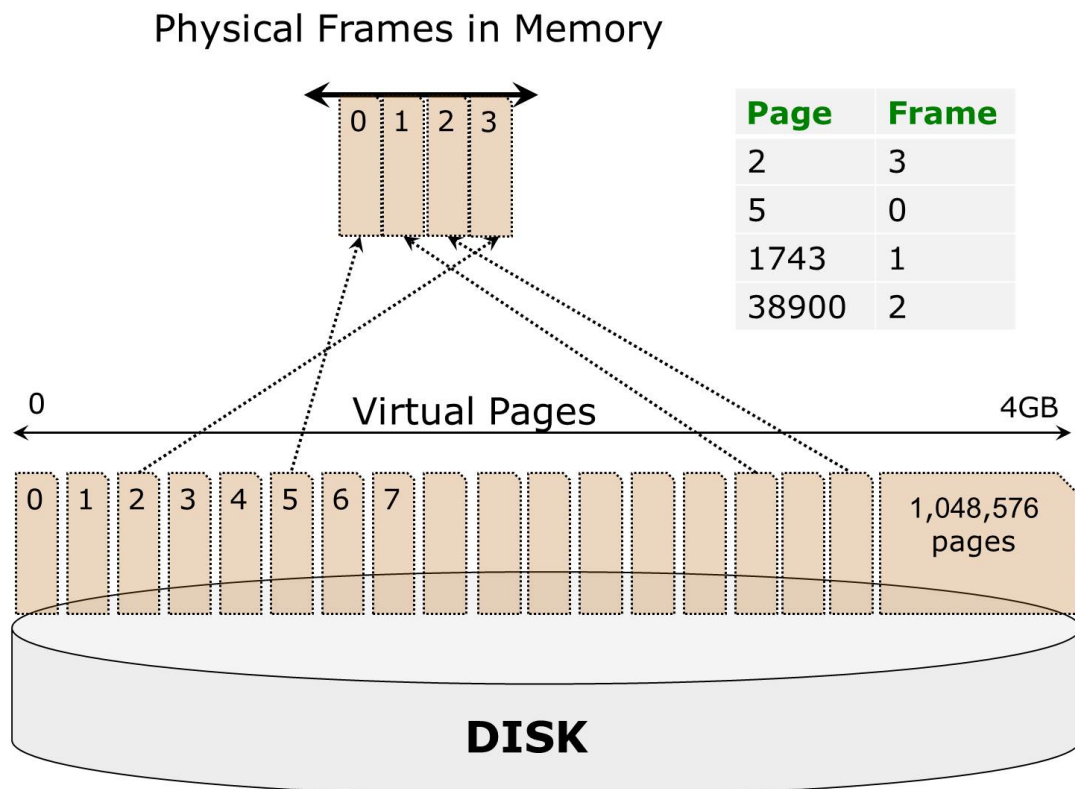
- אם למשל דף 2 (page2) ממופה למסגרת 17 (Frame17), אז הכתובת הוירטואלית $v = 11400$ (שנמצאת בדף 2) תתורגם לכתובת.

$$a = 17 * 4096 + 11400 \% 4096 = 72857$$

- אם גודלו של דף הוא חזקה של 2 (כמו למשך הגודל השכיח 4096), אז כל כתובת וירטואלית כגון $v = 11400$ ניתן להציג על ידי שני חלקים.

$$\begin{aligned}
 v &= 11400 \\
 &= 0000000000000000000010110010001000 \\
 &= \underbrace{0000000000000000000010}_{20 \text{ bits}} : \underbrace{110010001000}_{12 \text{ bits}}
 \end{aligned}$$

כאשר החלק הראשון (20 סיביות) מייצג את מספר הדף, בעוד שהחלק השני (12 סיביות) מייצג את הקיזוז מתחילת הדף (off-set).



איור 7.5: מרחב כתובות פיזי מול וירטואלי

- במידה והדף המבוקש אינו מופיע בטבלה, כלומר מדובר בדף חדש שנמצא בדיסק ואינו טעון בזיכרון, מתבצעת פסיקה מסוג Page Fault (פסיקת תוכנה מספר 14) אשר באמצעותה הדף החדש נטען לזכרון, טבלת הדפים מתעדכנת בהתאם, והתהליך חוזר לרוץ מהמקום בו הופסק.

- טבלת הדפים ממוקמת בזיכרון הראשי (לרוב ברמת cache הקרובה ביותר למעבד) ומנוהלת באופן יעיל על ידי מערכת ההפעלה לשם האצת מהירות הגישה לזיכרון. המערכות בהן טבלת הדפים גדולה מאוד, ניתן לשמור אותה גם כן בזיכרון הוירטואלי (כך שחלק מהטבלה עשוי להישמר בדיסק).

● MMU - Memory Management Unit

– עקב החשיבות הקריטית של מהירות הגישה (כתיבה או קריאה) לזיכרון, בשלב מסוים נוספה למעבד יחידת ניהול זיכרון יעודית. MMU (Memory Management Unit) אשר מטרתה להאיץ את מהירות פעולת התירגום מכתובת וירטואלית לכתובת פיזית.

– יחידת ה-MMU נעזרת בזיכרון מטמון אסוציאטיבי מיוחד שנקרא TLB (Translation Lookaside Buffer) ובמנגונוני חומרה יעודיים לתירגום מהיר מכתובת וירטואלית לכתובת פיזית.

– יחידת ה-MMU מחזיקה רק חלק קטן אך חשוב ביותר של

- טבלת הדפים (בו נעשה השימוש השכיח ביותר).
- כאשר הכתובת הוירטואלית שאנו מעוניינים לתרגם נמצאת ב-TLB (TLB hit), מהירות התירגום של יחידת ה-MMU קרובה למהירות הגישה הישירה לזיכרון.
 - הגישה ליחידת ה-MMU מהירה יותר מהגישה לזיכרון, ולכן ריבוי השימוש ביחידה זו מונע שימוש חוזר ויקר של גישה לזיכרון.
 - במידה והכתובת הוירטואלית אינה נמצאת ב-TLB, מתרחש ליקוי TLB (TLB miss), ומערכת ההפעלה עוברת לטבלת הדפים הראשית שבזיכרון.
 - לאחר תירגום הכתובת, מערכת ההפעלה תעדכן בהתאם את טבלת ה-TLB שביחידת ה-MMU. העידכון עשוי לכלול כתובות נוספות שנמצאות בקירבת הכתובת המקורית (אשר עשויות להופיע בבקשות תירגום עתידיות).
 - באופן כזה יחידת ה-MMU נשארת תמיד מעודכת עבור הכתובות הפעילות ביותר.

● טבלת דפים הירארכית (Multilevel Page Table)

- טבלת דפים רגילה עשויה להגיע לממדי ענק, ולכן בשלב מסוים נהגה רעיון הטבלה ההירארכית.
- מדובר למעשה במערכת של טבלאות קטנות בהרבה שמאורגנות בצורה הירארכית.

– תירגום של כתובות וירטואלית ייתבצע על ידי כניסה לטבלת דפים ראשית (root table). הטבלה הראשית תפנה אותנו לטבלת מישנה חדשה וכך הלאה עד להגעה לטבלת דפים רגילה שבאמצעותה נמצא את המסגרת (frame) המתאימה.

– כל רשומה בטבלה הראשית מצביעה לטבלת משנה נפרדת, ולכן היא אמורה להיות טבלה קטנה במיוחד.

– **דוגמא:** במערכת של 32 סיביות, עם גודל דף $4K$, ניתן לחלק את הטבלה הרגילה ל-1024 טבלאות קטנות, אשר בכל אחת מהן יש 1024 כניסות. הטבלה ההירארכית הראשית תכיל 1024 כניסות עבור 1024 טבלאות דפים רגילות. כתובת וירטואלית כגון $v = 56789000$, תתחלק לשלושה חלקים באופן הבא:

$$\begin{aligned}
 v &= 56789000 \\
 &= 00000011011000101000100000001000 \\
 &= \underbrace{0000001101}_{10 \text{ bits}} : \underbrace{1000101000}_{10 \text{ bits}} : \underbrace{100000001000}_{12 \text{ bits}}
 \end{aligned}$$

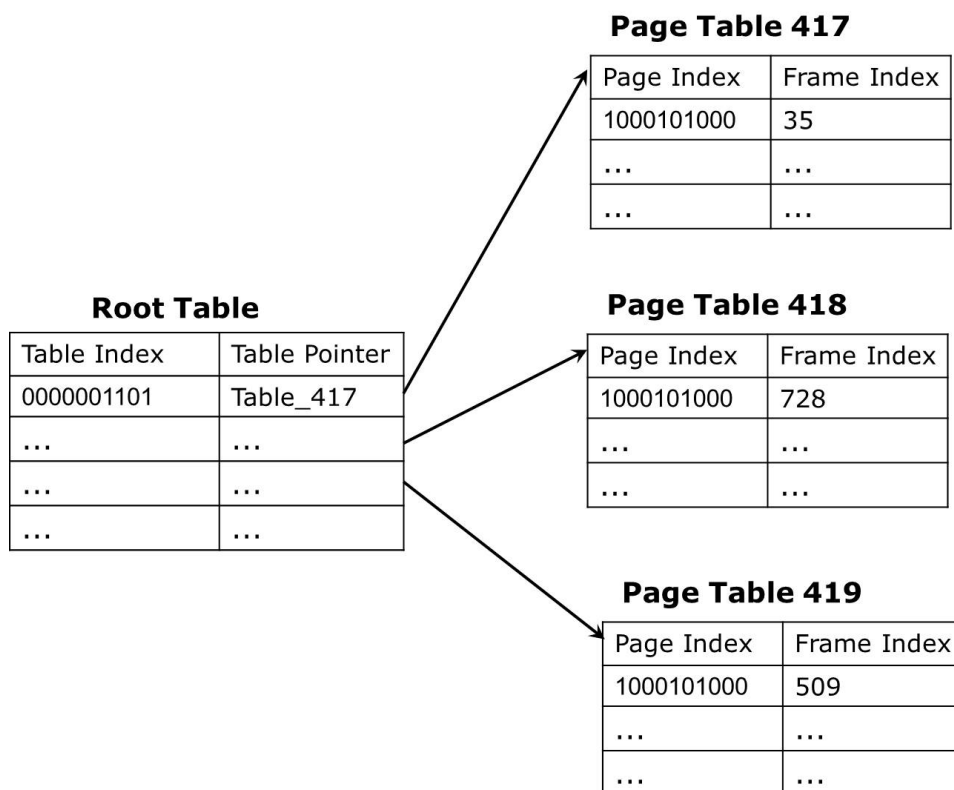
– החלק הראשון 0000001101 (10 סיביות) מציין את **מספר הטבלה המשנית אליה יש לפנות** (מתוך 1024 טבלאות אפשריות).

– החלק השני 1000101000 (10 סיביות) מציין את הכניסה (מספר דף) לטבלת הדפים המשנית.

- החלק השלישי 100000001000 (12 סיביות) מציין את הקיזוז (offset) של הכתובת הפיזית מתחילת המסגרת.
- הכתובת הפיזית תתקבל על ידי חיבור הקיזוז לבסיס המסגרת שמתקבלת בשלב השני

$$m = \text{frame}(n, p) * 4K + \text{offset}$$

הביטוי $\text{frame}(n, p)$ מציין את המסגרת שמתקבלת בטבלה n , כניסה p . הכתובת הפיזית מסומנת m .



איור 7.6: מרחב כתובות פיזי מול וירטואלי

7.4 מדיניות פינוי דפים

- בכל שלב בו כמות המסגרות הפנויות בזיכרון הפיזי יורד מתחת לסף מסוים, על מערכת ההפעלה להעביר חלק מהדפים לדיסק בכדי לפנות מקום לדפים חדשים.
- האתגר העיקרי שעומד בפני מערכת ההפעלה הוא למזער את מספר הפינויים ככל שניתן. פינויים תכופים מדי עשויים להאט את פעולת המערכת באופן משמעותי מאוד, ובמקרים מסוימים אף לגרום לקריסת המערכת כתוצאה מאי-עמידה בכמויות הדפים שיש להעביר (Thrashing)
- למטרה זו נעשו מאמצים רבים למציאת אלגוריתמים יעילים המסוגלים לנבא את הדפים שיש להוריד לדיסק (מאחר והסיכוי שיעשה בהם שימוש כלשהו בטווח הקצר קטן מאוד)
- דף אידיאלי לפינוי מהזיכרון אל הדיסק הוא דף שלא יעשה בו שימוש נוסף לעולם (או לפחות לא בטווח הנראה לעיין). אך דבר כזה לא ניתן לקבוע מראש בוודאות מוחלטת. המיטב שנוכל לעשות הוא לנבא זאת ברמת וודאות סבירה על בסיס נתונים שונים שנאגרו במהלך חייו של התהליך.
- הטבלה הבאה כוללת מספר שדות טיפוסיים שנכללים על ידי טבלאות דפים במערכות הפעלה שונות, אשר בין השאר משמשים חלק מהאלגוריתמים לפינוי דפים:

Page	Frame	Readonly	Modified	Valid	Reference	Protection	Age
0	38	0	0	1	45	user	431
1	15	1	0	1	0	user	152
2	617	0	0	1	14	user	23
3	31	0	1	1	32	user	658
4	548	1	0	1	26	kernel	72
5	9	0	0	1	8	user	60
6	83	0	0	1	0	kernel	23

מספר דף וירטואלי	Page	–
מספר מסגרת בזיכרון הפיזי	Frame	–
האם הדף שונה?	Modified	–
האם הדף בתוקף? האם הוא טעון לזיכרון הפיזי?	Valid	–
מספר גישות לכתובות בדף	Reference	–
הגנה ברמת משמש (user או kernel)	Protection	–
גיל הדף. הזמן שעבר מטעינתו לזיכרון הפיזי	Age	–

- בנוסף לשדות הנ"ל קיימים שדות נוספים שלא הזכרנו.
 - למשל בסביבות מסוימות קיימת סיבית שקובעת שדף מסוים לעולם לא יוסר מהזיכרון הפיזי.
 - שדה חשוב נוסף שלפעמים נלקח בחשבון הוא ההיסטוריה של מספר הפעמים שבו הדף פונה מהזיכרון ונטען שוב.
 - במקרים מסוימים השדה "Time of Last Reference" חשוב יותר ממספר הגישות
 - שדות אלה חשובים עבור אלגוריתמים למציאת הדף המתאים ביותר לפינוי במצבים של מצוקת זיכרון.

- זמן הטיפול בליקויי דף (page faults) בהם מעורב פינוי של דפים מלוכלכים (dirty), צורך זמן יקר מאוד. בכדי לא לעכב את תהליך המשתמש, פינוי דפים וכתובה של דפים "מלוכלכים" חזרה לדיסק נעשה באופן תדיר ברקע על ידי מערכת ההפעלה (באמצעות חוט או תהליך גרעין הממונה על משימה זו ופועל ללא הפסקה). ביצוע משימה זו ברקע, מאפשר כתיבה מיידית למסגרת שהתפנתה ללא צורך לחכות לפעולת כתיבה.
 - האלגוריתם הנפוץ ביותר לאיתור דפים מועמדים לפינוי נקרא אלגוריתם LRU (Least Recently Used) שמשמעותו היא לפנות את הדפים שבהם נעשה השימוש המועט ביותר לאחרונה.
 - זהו למעשה איפיון כללי מאוד המשמש בסיס למשפחה גדולה של אלגוריתמים שמיישמים רעיון זה, מאחר והמונח "נעשה שימוש מועט לאחרונה" ניתן לפרשנויות רבות, בהקשרים שונים.
 - דוגמא פשוטה לאלגוריתם מסוג זה תתקבל על ידי צפיה בעמודה Reference שבטבלה למעלה. שדה זה מציין את מספר הפעמים שבו התהליך ניגש לכתובות שונות בדף. הדף בעל ערך ה-Reference הנמוך ביותר הוא על כן המועמד הטוב ביותר לפינוי.
- אולם מספר זה עשוי להטעות אם כל הגישות האלה התבצעו בדיוק לאחרונה, ושלמעשה הדף רק התחיל את פעילותו.
- ייתכן גם שהדף בעל כמות הגישות הכי גדול השיג אותם

על ידי לולאה סתמית בעבר הרחוק, והדף למעשה שוכב לו בזיכרון בחוסר מעש מוחלט כבר הרבה זמן.

– ייתכן שפרמטר Time of Last Reference יהיה עדיף על פני מספר הגישות?

– דוגמא זו מדגימה יפה את הקשיים בבחירת האלגוריתם המעשי ביותר עבור בעיית פינוי הדף.

7.5 הערות מסכמות

- ניהול דפי זיכרון מעמיס על מערכת ההפעלה אחריות לא פשוטה מעבר לכל המטלות הרבות והמורכבות שמוטלות עליה.
- על מערכת ההפעלה מוטל לנהל טבלאות נתונים לגבי טבלת המיקומים הפיזיים של דפי הזיכרון
 - מהם הדפים שנטענו לזיכרון?
 - איפה נמצאים שאר הדפים שלא נטענו לזיכרון?
 - כיצד להגיע לדף כלשהו במידת הצורך?
 - ומהי הדרך היעילה ביותר להגיע לדף ספציפי?
- שיחלוף דפים בין הזיכרון לדיסק חייב להיות פעולה נדירה שמתבצעת לעיתים רחוקות מאוד. בנוסף לכך, כשפעולת השיחלוף מתרחשת, היא חייבת להיות מהירה מאוד. אחרת,

קצב התקדמות התהליך יואט באופן דראסטי, וכל עניין הדיפדוף לא יהיה מעשי.

- שיטת הדפים פותרת את בעיית הקיטוע (Fragmentation) שהיתה קיימת בשיטות הקודמות (כגון: segmentation) אשר בהן הזיכרון סבל ממספר גדל והולך של "חורים" ועקב כך חלק גדול ממנו לא יכל להיות מנוצל. בשיטת הדפים, בכל פעם שמסגרת זיכרון מתפנה, ניתן לאכלס בה מייד דף חדש בשלב הבא, כך שבכל שלב של התהליך, כל חור שנוצר ימולא מייד. זה כמובן מתאפשר מאחר וסדר הדפים בזיכרון הפיזי אינו חייב להתאים לסדר שלהם בזיכרון הוירטואלי.

- בניגוד לנאמר בתחילת הפרק, אחת התוצאות המפתיעות של שיטת הדפים היא האפשרות (הפשוטה מאוד לביצוע) שמספר תהליכים ישתפו ביניהם דפים ובכך להימנע מכפילויות, ולחסוך ביזבוז אדיר של זיכרון (כפי שזה קרה בתקופה הקודמת ל-paging).

– ספריות משותפות DLL (Dynamically Linked Library), כפי שהן נקראות במערכת ההפעלה Windows, או Shared Libraries כפי שהן נקראות במערכת ההפעלה Unix (או Lin-ux) נטענות על ידי התהליך בזמן הריצה (במקום להתקמפל עם התוכנית באופן סטטי)

– במידה וספריה משותפת נטענה קודם לכן על ידי תהליך

מסוים, מערכת ההפעלה תשתף את הדפים שלה עם כל תהליך נוסף שיבקש לטעון את אותה ספרייה, ובכך יחסכו המון כפילויות והמון זיכרון (ניתן לראות זאת בצורה ברורה על ידי הפעלת הפקודה `top` במערכת ההפעלה Linux).

– יתרון ברור נוסף לגישה זו הוא שאם מתגלה בעייה בספרייה, יש צורך לשחרר רק גירסא מתוקנת הספרייה עצמה. כל התוכנות שמשמשות בספרייה זו לא ייצטרכו לעבור שום שינוי עקב כך, וימשיכו לטעון את הגירסה המחודשת של הספרייה כמו קודם ללא שום שינוי.

– לבסוף נציין כי דפים של ספריות מערכת מוגנות בקפידה על ידי מערכת ההפעלה. הן בדרך כלל במצב `Readonly` ושייכות ל-`Kernel`, ולתהליכי המשתמש אין שום דרך לשנות אותם. לכן, שיתוף הדפים מסוג זה אינו בסתירה לעקרון אי-הפלישה של תהליך למשנהו.

– מערכת ההפעלה Linux ממשת את קריאת המערכת `fork()` על ידי שיכפול טבלת הדפים בלבד ולא שיכפול מלא של כל הזכרון של התהליך האב. השיכפול הפיזי מתבצע רק אם אחד מהתהליכים מחליט לכתוב על דף מסוים.

פרק 8

מעבד הפקודות Bash

- מעבד הפקודות הראשון נקרא בקצרה sh, ונכתב על ידי Steve Bourne בשנת 1979. למרות גילו, עדיין קיימות תוכניות רבות (בעיקר סקריפטים) של Unix המתבססות עליו.
- מאוחר יותר פותחו מעבדי פקודות יותר ויותר משוכללים כגון .bash, ksh, zsh, tcsh, csh.
- השכלול התבטא בהוספת כלים
- חלק מהמעבדים החדשים לא שמרו על תאימות עם המעבד הראשון - csh, tcsh.
- מסיבות מובנות רצוי שתוכניות שנכתבו במעבד הראשון ימשיכו לרוץ ללא שינוי גם במעבדים החדשים. מאמץ כזה נעשה בפיתוח .bash, ksh, המעבדים הנפוצים ביותר כיום.
- Bourne Again Shell - Bash

המעבד Bash פותח גם בכדי לעמוד בכל דרישות התקן Posix-1003.2. בנוסף לכך הוא מסופק חינם לכל דורש על ידי GNU (Free Software Foundation).

- **Bash** - מעבד הפקודות הסטנדרטי של Linux.

- המעבד ksh לעומת זאת הוא תוכנה מסחרית.

- מאפיינים עקריים של מעבדים אלה:

- הפעלה של סקריפטים

- אופרטורים להרצת תהליכים ברקע או בחזית. אפשרות לעבור בקלות מהרקע לחזית ולהיפך.

- loops, if/then, case, ...

- משתנים, פרמטרים, פונקציות.

- משתנים סביבתיים, משתנים מקומיים, והורשת הסביבה לתהליכים ילדים.

- שלושה אופני המרה.

- ריבוי רמות: אפשרות להפעיל מעבד אחד מתוך מעבד שני.

- אפשרות ללכידת סיגנלים.

- קובצי איתחול: `.bashrc`, `.bash_profile`, `.bash_history`, `./etc/bashrc`

- אפשרות לערוך פקודות שורה (command line), לחזור בקלות לפקודות שורה ישנות ולהריץ אותן לאחר עריכה (history).

● שני אופני הפעלה: interactive או noninteractive

- במצב אינטראקטיבי המעבד קולט פקודות מהמשתמש דרך מקלדת המסוף, וכותב את הפלט למסוף. זהו המצב השכיח, וכל משתמש נמצא בו מייד לאחר הכניסה למערכת.

- במצב לא-אינטראקטיבי המעבד קולט את הפקודות מתוך קובץ וכותב את הפלט לקובץ.

● הקלט של מעבד הפקודות מורכב מסדרת **פקודות**. כל פקודה יכולה להשתרע על פני מספר שורות.

● כאשר המעבד רץ באופן אינטראקטיבי, מיד לאחר סיום כל משימה הוא יציג **סימן מנחה** (prompt sign) כגון \$ בכדי לאותת למשתמש שהוא מוכן לקבל את הפקודה הבאה.

● קיים גם סימן מנחה משני (PS2) למקרים בהם פקודה ממשיכה לשורה הבאה.

● כל פקודה שהמעבד קולט עשויה להשתייך לאחת מהקבוצות הבאות:

א. פקודות פנימיות של המעבד כגון `cd`, `echo`, `exit`, `kill`, `pwd`, `logout`, וכו'.

ב. יישומים כגון `vi`, `ls`, `cp`, `rm`, `bash`, `ksh`, וכו'.

ג. סקריפטים (batch files) של המערכת או שמשמש עצמו כתב.

● הבדיקה של כל פקודה מתבצעת על פי הסדר הנ"ל.

- פקודות מקבוצות ב' או ג' מועברות לגרעין לביצוע.

- לשם הפעלת פקודות בקבוצה ב' המעבד משתמש במשתנה הסביבתי PATH ומעביר לגרעין את המסלול המלא של הפקודה (לפונקציה `exec`).

- במידה והפקודה משתייכת לקבוצה ג', הגרעין יבחר במעבד פקודות המוגדר לשם כך מראש (ברירת המחדל בדרך כלל היא `/bin/sh`) בכדי להריץ את הסקריפט, אלא אם כן השורה הראשונה של הסקריפט מציינת אחרת:

```
#!/bin/bash
script ....
#!/bin/ksh
script ....
```

8.1 תהליך העיבוד

● כל פקודה עוברת שני שלבים של עיבוד: בשלב הראשון הפקודה מתפרקת לסדרה של מילים (tokens). אלה הן צרופי תווים אשר

המעבד מתייחס אליהם כיחידות תחביריות יסודיות. בשלב השני המעבד מארגן את כל המילים האלה לפקודה שלמה בעלת מובן לגרעין. היא מועברת לאחת מפונקציות הגרעין (exec). הפרטים הכרוכים בשלב העיבוד השני יפורטו בהמשך (בקצרה: פיתוח ביטויים רגולריים, המרת משתנים, ועוד).

- קיימים שני סוגים של מילים: מחרוזות תווים רגילות ואופרטורים. מחרוזת היא רצף של תווים השונים מהתווים space, tab, newline, או תווי אופרטורים. המחרוזת מופרדת על ידי תווי space, tab, newline, אלא אם תווים אלה נמצאים תחת ציטוט.

- האופרטורים הם

```
>  >>  >&  >|  <  <<  <<-  <&  <>
|  &  ;  (  )  &&  ;;  ((  ))  |&
```

- דוגמאות למחרוזות:

```
abcdef  vi  ls  ab*.*  PATH=$PATH:/sbin
"abc def gy*"  abc\ def\ gy*.txt  'abc $def'-text
'who am i'  $(finger u3204318)
```

- האופרטורים בשורה הראשונה הם אופרטורים להפניית קלט או פלט. ניתן לצרף אליהם את הספרות 0, 1, או 2 בכדי לציין את מקור הקלט או הפלט. בכדי שזה יהיה ברור למעבד, הספרה צריכה להיות צמודה לסימן האופרטור, והצרוף הזה צריך להיות מופרד על ידי רווחים משני הצדדים.

```

cat myfile 2> yourfile
=> Redirect stderr to yourfile
cat myfile2> yourfile
=> This is interpreted as:
cat myfile2 > yourfile

```

- המשמעות של תו newline תהיה תלויה בהקשר שלו:

א. אם הוא בא מייד לאחר פקודה **שלמה**, הוא יסיים את הפקודה, ובכך הוא זהה לתו ;

ב. אם שורת הפקודה ריקה, המעבד יתעלם ממנו.

ג. אם הפקודה אינה שלמה הוא ישמש כרווח רגיל, והשורה הבאה תמשיך את הפקודה.

- **ציטוט**: space, tab, או newline המופיעים בתוך ציטוט אינם נחשבים למפרידים והם יהוו חלק מהציטוט עם המשמעות הרגילה שלהם.

- ציטוט יוצר מילה אחת בלבד, למרות שחלק מהתווים המופיעים בציטוט הם תווים "לבנים". באופן דומה ציטוט משורשר לסדרת תווים לא לבנים, או שירשור של שני ציטוטים יוצרים מילה אחת.

```

TMP="This is a very long name"
vi $TMP.txt
=> The shell first makes a substitution
vi This\ is\ a\ very\ long\ name.txt
=> Next level: the shell sends
exec(/bin/vi, /home/user27/This is ...txt")
=> to the kernel.

```

```
export PATH1=/bin:/usr/bin:/usr/X11R6/bin
export PATH2=~ /bin:/home/cs/stud97/user65/bin
export PATH=$PATH1:$PATH2
```

● שלושה סוגי המרה:

- המרת פקודה - command substitution
- המרת פרמטר - parameter expansion
- חישוב ביטוי אריתמטי - arithmetic evaluation

- בסוף כל המרה מתקבלת מחרוזת תווים אשר המעבד מתייחס אליהן כאילו הוקלדו ישירות על ידי המשתמש. כלומר, לתווים לבנים ולאופרטורים תהיה המשמעות הרגילה.

- **הערות**
הערה מתחילה בתו #, בתנאי שהוא מתחיל מילה חדשה. ההערה תסתיים בסוף השורה. הערות הן חשובות מאוד עבור סקריפטים.

```
#This is a good comment
sort /etc/passwd # good comment
echo This is not#good
```

8.2 פקודות פשוטות

- פקודה פשוטה היא פקודה פנימית של המעבד או קריאה רגילה לתוכנית יישום. מעבד הפקודות bash מאפשר גם לתת שמות נוספים לפקודות באמצעות הפקודה alias.

- תחביר של פקודה פשוטה:

- מילה ראשונה היא שם של תוכנית, פקודה פנימית, או `alias`.
- שאר המילים הם ארגומנטים.
- שום ארגומנט אינו אופרטור.

- אפשר לאתחל משתנים לפני הפקודה

```
PATH=/home/mybin OPTS="+-" mycmd args
```

- סדר חיפוש פקודה

1. ראשית כל המעבד יבדוק אם שם הפקודה הוא אחד

מהמילים השמורות

```
!      elif      fi      in      while
case  else      for     then   {
do    esac      if      until  }  done
```

2. האם הפקודה היא `?alias`

אם כן, היא מוחלפת בשם הישן. הערך שמתקבל נבדק שוב מהתחלה (הוא עשוי להיות מילה שמורה!). אם גם הוא `alias`, התהליך חוזר על עצמו בצורה רקורסיבית.

3. האם הפקודה היא **פקודה פנימית** של המעבד?

קיימים בערך 50 פקודות פנימיות:

```
:      .      source  alias   bg      bind    break
builtin cd      command continue declare
typeset dirs   echo    enable  eval
exec   exit   export  fc      fg      getopt
hash  help   history jobs    kill    let
```

```

local      logout   popd       pushd      pwd        read
readonly  return   set        shift     suspend
test       times    trap       type      ulimit    umask
unalias    unset     wait

```

העובדה שפקודות אלה פנימיות מבטיחה שהן לרשות המשתמש כמעט בכל מצב.

שאלה: איזו בעייה היתה עשויה להתעורר אם הפקודות exit או logout למשל לא היו פנימיות?

4. האם הפקודה היא פונקציית מעבד (שהוגדרה על ידי המערכת או המשתמש)?
דוגמאות להגדרת פונקציות:

```

ls() {
    /bin/ls -FC -s "$@" | less
}
myfind() {
    find $HOME -name $1 -print
}
locate() {
    /usr/bin/locate "$@" | less
}

```

הגדרות אלה נמצאות בדרך כלל באחד מקובצי האיתחול של bash.

5. האם הפקודה היא שם אבסולוטי (מסלול) של תוכנית יישום?

מאחר ושם אבסולוטי מתחיל בתו / מקרה זה מבטל את כל המקרים האחרים.

6. האם הפקודה היא תוכנית יישום או סקריפט שניתן להגיע

אליו באמצעות המשתנה הסביבתי PATH?

- אם התשובה לכל השאלות הנ"ל היא שלילית, אז הפקודה אינה קיימת. המעבד יימסור הודעת שגיאה למשתמש.

- אינפורמציה מלאה על הפקודות הפנימיות: `man bash`.

- תרגיל: קרא את דף העזרה של `bash` ולמד כיצד להשתמש בפקודות הפנימיות הבאות

```
.          source  alias  bg      fg
builtin  typeset  export hash   help
history  jobs      kill   logout
popd     pushd    type   ulimit
umask    unalias  unset
```

- דוגמאות לקיצורים שימושיים במיוחד:

```
alias k9='kill -9'
alias cdrom='mount -t iso9660 -r /dev/cdrom /mnt/cdrom'
alias dir='ls -sF'
alias md=mkdir
alias rd=rmdir
alias copy='cp -i'
alias rename='mv -i'
alias ren=rename
alias del=rm
alias deltree='/bin/rm -R -f'
```

מומלץ ליצור קובץ נסתר עם שם כמו `.bash_aliases`, אשר יכלול בתוכו את כל הקיצורים של המשתמש, ולהפעילו מתוך הקובץ `.bashrc` או `.bash_profile`. על ידי הפקודה:

```
source ~/.bash_aliases
```

8.3 קישור פקודות בעזרת אופרטורים

- האופרטור `|` מקשר בין שתי פקודות באמצעות `pipe`:

```
cmd1 | cmd2
```

הפלט הסטנדרטי של הפקודה `cmd1` הופך להיות הקלט הסטנדרטי של הפקודה `cmd2`. ניתן לקשר יותר משתי פקודות:

```
ps aux | grep rony | more
cat /etc/passwd | sort | less
ls -a -R -l | grep eggdrop | less
cat /home/cs/stud97/*/.bash_history | grep "cmd"
```
- `cmd1 || cmd2`

אם קוד היציאה של `cmd1` שונה מאפס (הפקודה לא התבצעה בהצלחה) אז תתבצע `cmd2`. במילים אחרות: הפקודה `cmd2` תתבצע אם ורק אם ביצוע הפקודה `cmd1` נכשל.
- `cmd1 && cmd2`

אם קוד היציאה של `cmd1` הוא אפס (הפקודה התבצעה

בהצלחה) אז תתבצע גם הפקודה cmd2. במילים אחרות: הפקודה cmd2 תתבצע אם ורק אם ביצוע הפקודה cmd1 הצליח.

• ;

זהו סימן סיום פקודה. דוגמא:

```
cp /etc/passwd /tmp ; cd /tmp ; vi passwd
```

כאשר המעבד פוגש בסימן ";" הוא יבצע את הפקודה שבאה לפניו ומייד לאחר מכן יעבור לפקודה הבאה.

• command &

הסימן & מציין כי הפקודה שלפניו צריכה להתבצע **ברקע**. כאשר מעבד הפקודות רואה את הסימן & הוא יפעיל את הפקודה שלפניו, לא יחכה לסיומה, ומייד יעבור לביצוע הפקודה הבאה (אם ישנה).

דוגמאות:

```
sort bigfile > /tmp/sorted_file &
sort file1 > file2 & vi file3
```

• סדר הקדימות של האופרטורים הנ"ל:

```
|
|| &&
; &
```

8.4 הפניית קלט ופלט

- `< file`

קח את הקלט הסטנדרטי מתוך הקובץ `file`.

- `command << [-] word`

קח את הקלט הסטנדרטי מתוך הטקסט הבא מייד לאחר המילה `word`. הקלט חייב להתחיל בשורה חדשה. סוף הקלט יסומן על ידי המילה `word` גם כן (בשורה נפרדת).
דוגמא לסקריפט הכולל את הקלט בתוכו:

```
sort <<ZZZ
aaaaa
sssss
cccc
ffffffffff
ccccccc
ZZZ
```

סוג כזה של הפניית קלט נקרא לרוב בשם `here-document`. שימושי במיוחד עבור יצירת קבצים דחוסים הפותחים את עצמם (על ידי הפעלתם כתוכנית).

הצורה "`<<-`" גורמת להתעלמות מתווים לבנים מהתחלת כל שורה.

- `command > file`

הפלט הסטנדרטי של הפקודה `command` יישלח לקובץ `file`. אם הקובץ `file` קיים, תוכנו הקודם יימחק, גם אם הפקודה לא תיצור שום פלט. אם אינו קיים, הוא נוצר.

- `command >> file`
אם הקובץ `file` קיים הפלט הסטנדרטי של הפקודה `command` ייתווסף לקובץ `file`. אם הקובץ `file` אינו קיים הוא נוצר.
- בתחילתו של כל תהליך, מערכת ההפעלה תפתח את שלושת הקבצים הסטנדרטיים הקשורים למסוף. בנוסף לקבצים אלה, התהליך עצמו עשוי לפתוח קבצים נוספים עבור פעילותו. מעבדי הפקודות החדשים מאפשרים למשתמש לפתוח קבצים בצורה פשוטה יחסית:

```
0  standard input      (stdin)
1  standard output    (stdout)
2  standard error     (stderr)
m< fname  open the file "fname" for reading
           with file-desc m
m> fname  open the file "fname" for writing
           with file-desc m
n<>file  open file for reading and writing
           its file-desc is n
m>> fname  open the file "fname" for appending
           with file-desc m
```

● דוגמאות:

```
cat 3<myfile <&3
same as:
3<myfile cat <&3
same as:
cat < myfile
same as:
cat myfile
3< file1 4>> file2 1>&4 cat <&3
```

- לאחר שברשותנו מספר קבצים פתוחים, נוכל בקלות לשנות את יעדיהם:

```
>&n file associated with file-desc n
will be associated with stdout
<&n file associated with file-desc n
will be associated with stdin
>&- close standard input
<&- close standard output
```

- בצורה יותר כללית:

```
m>&n file-desc m points to the same input file
with file-desc n
m<&n file-desc m points to the same output file
with file-desc n
m>&- close input file with file-desc m
m<&- close output file with file-desc m
```

- דוגמאות:

```
ls > dirlist 2>&1
standard output and standard error
of the command ls
goes to the file dirlist
ls 2>&1 > dirlist
This is not the same!
Only stdout(1) is going to the
file dirlist.
Stderr(2) was duplicated as
stdin, so it does not exist.
cmd 3>&1 1>&2 2>&3 3>&-
Standard output becomes standard error
and standard error becomes standard output
file descriptor 3 is not needed
```

8.5 הפקודה test

- הכוונה לפקודה הפנימית test של מעבד הפקודות bash.
- במערכות Unix שונות קיימת לפעמים פקודה חיצונית באותו שם שאינה קשורה לפקודה שלנו. יש להזהר מבילבול ביניהן.
- מטרת הפקודה היא לבצע בדיקות שונות כגון:
 - השוואה בין מחרוזות
 - האם מחרוזת נתונה היא ריקה?
 - קיום או אי-קיום קובץ
 - האם קובץ מסוים הוא קובץ רגיל או ספרייה?
 - האם קובץ הוא ריק?
 - השוואה בין תאריכי קבצים
 - השוואה בין מספרים שלמים
- `test expr`

הפקודה מקבלת ארגומנט אחד *expr* שהוא ביטוי בוליאני ותחזיר 0 (true) או 1 (false), בהתאם לערך של הביטוי הבוליאני *expr*.

קיימת צורה שקולה של פקודה זו: `[expr]`
- תאור הבוררים הקשורים לקבצים:

`test expr`

```
[ expr ]
```

Return a status of 0 (true) or 1 (false) depending on the evaluation of the conditional expression `expr`. Expressions may be unary or binary.

Unary expressions are often used to examine the status of a file.

If file is of the form `/dev/fd/n`, then file descriptor `n` is checked.

`-b file`

True if file exists and is block special.

`-c file`

True if file exists and is character special.

`-d file`

True if file exists and is a directory.

`-e file`

True if file exists.

`-f file`

True if file exists and is a regular file.

`-g file`

True if file exists and is set-group-id.

`-k file`

True if file has its sticky bit set.

`-L file`

True if file exists and is a symbolic link.

`-p file`

True if file exists and is a named pipe.

`-r file`

True if file exists and is readable.

`-s file`

True if file exists and has a size greater than zero.

```
-S file
    True if file exists and is a socket.
-t fd
    True if file-desc fd is opened on a terminal.
-u file
    True if file exists and its set-user-id bit
    is set.
-w file
    True if file exists and is writable.
-x file
    True if file exists and is executable.
-0 file
    True if file exists and is owned by the
    effective user id.
-G file
    True if file exists and is owned by the
    effective group id.
file1 -nt file2
    True if file1 is newer (modification date)
    than file2.
file1 -ot file2
    True if file1 is older than file2.
file1 -ef file2
    True if file1 and file2 have the same device
    and inode numbers.
```

• דוגמאות

```
if test -e .bashrc; then echo "It exists!"; fi
if test file1 -nt file2; then cp file1 file2; fi
if [-x myfile]; then echo "Is executable" ; fi
This is the same as:
if test -x myfile; then echo "Is executable" ; fi
if [-e myfile -a -x myfile]; then source myfile; fi
```

• תאור הבוררים הקשורים למחרוזות וחשבון:

`-z string`

True if the length of string is zero.

`-n string`

`string`

True if the length of string is non-zero.

`string1 == string2`

True if the strings are equal.

`string1 != string2`

True if the strings are not equal.

`! expr`

True if expr is false.

`expr1 -a expr2`

True if both expr1 AND expr2 are true.

`expr1 -o expr2`

True if either expr1 OR expr2 is true.

`arg1 OP arg2`

OP is one of `-eq`, `-ne`, `-lt`, `-le`, `-gt`, `-ge`.

These arithmetic binary operators return

true if arg1 is equal, not-equal, less-than,

less-than-or-equal, greater-than,

or greater-than-or-equal than arg2.

Arg1 and arg2 may be positive integers,

negative integers, or the special

expression `-l string`, which evaluates to the

length of string.

8.6 פקודות מורכבות

- פקודה מורכבת היא פקודה שנוצרת על ידי צרוף של פקודות פשוטות באופנים שונים.

- הצרוף מתבצע באמצעות המילים השמורות הבאות:

```
if then else elif fi
case in esac
for while until do done
break continue
```

- המעבד מזהה מילים אלה רק במקומות בהן פקודות רגילות אמורות להופיע: התחלת שורת פקודה, לאחר התו ";", לאחר האופרטורים || או &&, וכדומה.

- הפניית קלט/פלט תוכל להתבצע גם לאחר פקודה מורכבת.

- **ביצוע מותנה - if**

```
if list
  then list
[elif list
  then list]
. . .
[else list]
fi
```

- המונח list מציין פקודה שבה עשויים להופיע האופרטורים: |, |, &, &&, ;, !

- המילים if, then, elif, fi חייבות להתחיל בשורה חדשה או לבוא מייד לאחר התו ";"

- קוד יציאה: אם כל התנאים נכשלים ואין else אז קוד היציאה הוא 0. בכל המקרים האחרים, קוד היציאה הוא זה של ה-list שמתבצע.

- **דוגמאות:**

הקטע הבא עשוי להופיע למשל בקובץ .bashrc, בכדי להפעיל קובץ קיצורים (aliases).

```
ALIASES=$HOME/.bash_aliases
```

```
if test -e $ALIASES
```

```
then source $ALIASES
```

```
fi
```

Different approach:

```
ALIASES=$HOME/.bash_aliases
```

```
if
```

```
test -e $ALIASES
```

```
then
```

```
source $ALIASES
```

```
echo "aliases ready"
```

```
else
```

```
echo "No aliases file"
```

```
fi
```

- **בדיקת מקרים - case**

```
case word in
```

```
  pat1) list1 ;;
```

```
  pat2) list2 ;;
```

```

. . .
patn) listn ;;
esac

```

- אם המילה word תואמת את אחת מהתבניות אז רק ה-list השייך לתבנית הראשונה שתואמת ייתבצע.

- קוד יציאה של כל הפקודה הוא קוד היציאה של המקרה שהתבצע. אם אף מקרה לא התבצע אז קוד היציאה הוא 0.

- **דוגמאות:**

```

case $fn in
*.c | *.cpp)  cppcompile $fn ;;
*.pas        )  pascal_compile $fn ;;
*.f77        )  fortran77 $fn ;;
*            )  echo "Not a source file!"
esac

```

```

case $HOSTNAME in
mail*)
echo "You are working on mail computer" ;;
moon*)
echo "You are working on moon" ;;
sun*)
echo "You are working on sun" ;;
esac

```

- **סריקת רשימת מחרוזות - for**

```

for name in names_list
do
list
done

```

● **דוגמאות:**

```
for user in miki uri liat dany rina
do
mail $user@braude.ac.il < message-file
echo "I have sent mail to $u"
done
for f in *.exe *.obj
do
echo -n "Remove $f? (y/n/q) "
read response
case $response in
y*|Y*)  rm  $f ;;
q*|Q*)  echo "Quitting!"
        break ;;
*)      echo "File $f not removed" ;;
esac
done
```

● **לולאת while**

```
while list
do
  list
done
```

● **ה-`list` הראשון הוא בדרך כלל פקודת `.test`**● **דוגמא:**

```
#!/bin/bash
N=1
while true
do
```

```
if
  test $N -eq 1
then
  echo "Go home! it is late!"
  N=2
elif
  test $N -eq 2
then
  echo "No CPU found! Stop work!!"
  N=1
fi
sleep 30
done &
```

• לולאת until

```
until list
do
  list
done
```

כמו `while` אך המבחן הוא הפוך: הלולאה תפעל כל עוד הפקודה `list` אינה מצליחה (קוד יציאה שונה מאפס). הלולאה תעצר בהזדמנות הראשונה שהפקודה `list` תתבצע בהצלחה (קוד יציאה אפס).

8.7 שלושה אופני ביצוע

- למעבד הפקודות `bash` יש שלושה דרכים שונות לביצוע פקודה (פשוטה או מורכבת):

- **ביצוע ישיר**

- **ביצוע ישיר בתת-מעבד**

- **ביצוע לא ישיר בתת-מעבד**

- ברוב דפי העזרה ההבחנה הזו אינה ברורה לחלוטין, אך חשוב מאוד להבין אותה בכדי לדעת אם משתנים סביבתיים או משתנים לוקליים עוברים בירושה הלאה.

● **ביצוע ישיר**

- הפקודה מתבצעת ברגע הראשון שבו המעבד פוגש אותה.
- כל שינוי הנגרם על ידי הפקודה למשתנים הסביבתיים או לקבצים הפתוחים יישארו בתוקף גם לאחר סיום הפקודה עד לסיום המעבד עצמו (או עד לביצוע ישיר של פקודה אחרת המשנה אותם).
- פקודות פנימיות של המעבד, aliases של המשתמש, ופונקציות של המשתמש, כולן מבוצעות באופן ישיר.

● **ביצוע ישיר בתת-מעבד**

- המעבד משתמש בפונקציית הגרעין fork בכדי ליצור עותק זהה של עצמו לפני ביצוע הפקודה. באופן כזה מתקבלים שני מעבדים שונים: המעבד המקורי ותת-מעבד.
- תת-המעבד מבצע את הפקודה.

- במידה ותת-המעבד עובד בחזית (foreground) אז המעבד הראשון נכנס למצב wait ומחכה לסיום תת-המעבד. כל התהליכים הנוצרים על ידי התת-מעבד, ולא סיימו לאחר שהתת-מעבד עצמו סיים, יהפכו אוטומטית לילדים של התהליך init (זהו תהליך שמספרו 1).
- אפשרות שניה: תת-המעבד עובד ברקע (background) והמעבד המקורי חוזר מייד לפעולה.
- בשני המקרים, תת-המעבד יורש מהמעבד הראשון את כל המשתנים הסביבתיים שהיו בתוקף לפני ביצוע הפקודה.
- כל שינוי, הוספה, או מחיקה של משתנים סביבתיים וכדומה המתבצעים על ידי התת-מעבד לא יישארו בתוקף לאחר סיומו. במילים אחרות, לאחר שתת-המעבד יסיים את ביצוע הפקודה, המשתנים הסביבתיים יהיו בדיוק כמו לפני ביצוע הפקודה.
- איזה פקודות מתבצעות בדרך זו?
 - א. פקודות בין סוגריים: (list)
 - ב. פקודות שמתבצעו ברקע: list &
 - ג. פקודות שמתבצעו בתוך ציטוט: 'list', \$(list)
 - ד. כל הפקודות המופיעות בתוך pipeline מלבד האחרונה.

● ביצוע לא ישיר בתת-מעבד

- המעבד יוצר תת-מעבד שהוא עותק של עצמו.

- תת-המעבד משתמש באחת מפונקציית הגרעין `exec` בכדי לבצע את הפקודה.
- התת-מעבד יורש את כל המשתנים הסביבתיים אך לא את המשתנים הלוקליים.
- לאחר ביצוע הפקודה מצב הסביבה חוזר לקדמותו גם אם תת-המעבד ביצע שינויים במשתנים הסביבתיים.
- איזה פקודות מתבצעות בדרך זו?
 - א. תוכניות יישום חיצוניות כגון `vi`, `mail` וכדומה.
 - ב. סקריפטים (shell scripts).

● הפקודה `export`

בכדי להפוך משתנה לוקלי למשתנה סביבתי שיעבור הלאה לתת-מעבדים יש להשתמש בפקודה `export`.

```
host> cd ~/dir1
host> X=1 Y=2 Z=3
host> export X
host> (cd ../dir2 ; pwd ; X=6 Y=7 ; echo $X $Y $Z)
/home/user62/dir2
6 7 3
host> pwd
/home/user62/dir1
host> echo $X $Y $Z
1 2 3
```

שים לב כי למרות שהמשתנה `X` שייך לסביבה הוא לא השתנה!
למה?

- בהמשך לדוגמא הקודמת, ננחח שאנו יוצרים קובץ בשם `tryit` שתוכנו הוא:

```
#!/bin/bash
cd ~/dir2
pwd
Y=7
echo $X $Y $Z
```

נבצע את הפקודות הבאות:

```
host> tryit
/home/user62/dir2
1 7
host> pwd
/home/user62/dir1
host> echo $X $Y $Z
1 2 3
```

- מה קרה למשתנה `?Z`

8.8 שלושה סוגי פרמטרים

- במעבד הפקודות `bash` ישנם שלושה סוגי פרמטרים:
 - פרמטרי מיקום (positional parameters)
 - פרמטרים מיוחדים (special parameters)
 - משתנים

- אם param הוא פרמטר אז \$param או \${param} הוא הערך שלו.

- **פרמטרי מיקום (positional parameters)**

בכל פעם שמפעילים סקריפט שמקבל ארגומנטים, המעבד מאתחל את פרמטרי המיקום באופן הבא:

```
$0 The name of the script
$1 Argument 1
$2 Argument 2
. . .
$n Argument n
```

- ניתן להשתמש בהם בתוך הסקריפט בכדי לטפל בארגומנטים של הסקריפט. במידה ומספר הארגומנטי עולה על 9 יש להשתמש בצורה \${17}.

- **דוגמא:**

```
host> whoop dog cat bird
$0 = whoop
$1 = dog
$2 = cat
$3 = bird
host> whoop "dog cat" bird
$0 = whoop
$1 = "dog cat"
$2 = bird
```

- הסקריפט הבא, שנקרא לו `comp`, מקבל שני ארגומנטים שהם שמות קבצים קיימים ועורך השוואה בין תאריכי השינוי האחרון שלהם:

```
host>comp readme.txt mymail.txt
comp: readme.txt is older than mymail.txt
-----The script-----
#!/bin/bash
file1=$1
file2=$2
if
  test ! -e $file1
then
  echo "$0: $file1 does not exist!"
elif
  test ! -e $file2
then
  echo "$0: $file2 does not exist!"
elif
  test $file1 -ot $file2
then
  echo "$file1 is older than $file2"
elif
  test $file1 -nt $file2
then
  echo "$file1 is newer than $file2"
else
  echo "$file1 has the same mtime as $file2"
fi
```

● משתנים

- משתנים קיימים בכדי לשמור על מידע חשוב במסגרת סקריפטים ולשם פעולה שוטפת של המעבד עצמו.
- בתחילת פעולתו של המעבד, קיימים כבר מספר משתנים (כגון PATH) המיוצאים (export) באופן אוטומטי לכל תהליך שנוצר על ידי המעבד. משתנים אחרים הם לוקליים, ואינם עוברים הלאה. בכדי ליצא משתנה לוקלי, יש להשתמש בפקודה export.
- שם של משתנה חייב להתחיל באות או בסימן " _ ". לאחר מכן מותר להשתמש באותיות ומספרים בכל סדר שהוא. קיימת הבחנה בין אותיות גדולות וקטנות.
- המשתנים הסביבתיים של המעבד עוברים תמיד לכל תהליך שמתבצע בתת-מעבד. המשתנים הלוקליים, לעומת זאת, עוברים רק לתהליכים שמתבצעים באופן ישיר בתת-מעבד. בכל מקרה, לאחר סיום התהליך, מצב המשתנים חוזר לקדמותו, גם אם התהליך ביצע שינויים (למעשה תת-המעבד מקבל עותק משלו של המשתנים, ולאחר סיומו עותק זה נמחק).

● הפקודות export, typeset, readonly

- לפקודה typeset יש עוד מספר תפקידים מלבד ייצוא משתנים. פקודות האלה הן פקודות פנימיות ולכן ניתן ללמוד עליהן רק דרך דף העזרה של bash (הפעל man bash).

`export var`

```
typeset -x var
                export the variable var
typeset +x var
                cancel export
typeset -r var
declare -r var
readonly var
                make var readonly
typeset +r var
declare +r var
                cancel readonly
```

● פרמטרים מיוחדים הקשורים לארגומנטים של סקריפט

מייד לאחר הפעלת סקריפט, המעבד (אשר בו הסקריפט מופעל) מאתחל שורה של פרמטרים חדשים הקשורים לארגומנטים שהועברו לסקריפט, אשר ניתן להשתמש בהם בתוך הסקריפט:

```
$0, $1, $2, ..., $n (discussed above)
$@ all the arguments together in a row
starting with $1. That is, "$@" is
equivalent to: $1 $2 ... $n
the spaces count as separators.
$* Same as @$ except that "$*" is
equivalent to "$1 $2 ... $n$" as
a single word!
$# number of arguments
$$ the process id of the current process
$- the flags to the script
$? the exit code returned by the most
```

```

recent executed foreground command
$! the exit code returned by the most
recent executed background command

```

8.9 שלושה אופני ציטוט

• ציטוט תווים (backslash quotation)

הצורה `\c`, כאשר `c` הוא תו כלשהו, מצטטת את התו `c` כמות שהוא באמת ומנטרלת כל פונקציה מיוחדת שעשויה להיות לתו זה במסגרת המעבד.

```

host> export VAR=this\ is\ a\ long\ quote
host> echo \$VAR=$VAR
$VAR=this is a long quote
host> VAR2=!\\(\*\)\&
host> echo $VAR2
!(*)&

```

• התו `\` בסוף שורה מבטל את התו `newline` שבא אחריו וממשיך את הפקודה לשורה הבאה.

```

host> echo This is a sin\
>gle line
This is a single line

```

• ציטוט פשוט (single quotation)

ציטוט פשוט של מחרוזת `string` מתבצע עלידי הוספת התו

' בתחילת ובסוף המחרוזת 'string'. ציטוט כזה מבטל את המשמעות המיוחדת של כל התווים המיוחדים

```
host> VAR='!(*)& \'
host>echo $VAR
!(*)& \
```

● ציטוט כפול (double quotation)

הצורה "string" היא צורה חלשה יותר של ציטוט אשר בה כל התווים מטופלים כתווים רגילים מלבד ארבעת התווים "\$", ",", "\", ' , המקבלים את המשמעות הבאה:

- התו " מסמן כמובן את סוף הציטוט, אלא אם הוא בא בצורה "\."

- התו \$ משמש לשם המרת משתנים. כלומר, ניתן להמיר משתנים כרגיל גם בתוך ציטוט כפול.

- התו ' משמש לשם המרת פקודת ביניים (הנושא יידון בפרק הבא). כלומר, ניתן לבצע המרת פקודת ביניים גם בתוך ציטוט כפול.

- התו \ מבטל את המשמעות המיוחדת של כל ארבעת התווים האלה.

- צרוף התווים \ ו-newline מתבטל יחדיו.

- כל הופעה של התו \ לפני תו שאינו אחד מארבעת התווים הנ"ל ואינו תו newline תותיר אותו במחרוזת.

● **דוגמאות:**

```
host> X='prot & gamb'
host> Y="The password of $X is '\$&*'"
host> echo $Y
The password of prot & gamb is '\$&*'
```

8.10 שלושה אופני המרה

- כזכור, מעבד הפקודות bash מאפשר שלוש צורות של המרה:
 - המרת פקודה - command substitution
 - המרת פרמטר - parameter expansion
 - חישוב ביטוי אריתמטי - arithmetic evaluation
- הסוג השני כבר נדון בהרחבה. לכן נדבר רק על הסוג הראשון והשלישי.
- **המרת פקודה (command substitution)**

מאפשרת להפעיל פקודה בתוך פקודה. הפלט של פקודת הביניים יהפוך להיות חלק של הפקודה העקרית. פקודת הביניים עשויה להיות גם פקודה מורכבת. קיימות שתי צורות שקולות: `$(command)` או `'command'`.

```
moon> id
uid=837(user41) gid=603(stud97)
moon> myid=$(id)
or
```



```
moon> myid='id'
moon> echo $myid
uid=837(user41) gid=603(stud97)
moon> grep user6 /etc/passwd
user6:A@vHj%G:861:603:Dan Raz:/home/stud97/user6:/bin/bash
moon> pwline=$(grep user56 /etc/passwd)
moon> MyName=$(grep user56 /etc/passwd | cut -d : -f 5)
moon> MyShell=$(grep user56 /etc/passwd | cut -d : -f 7)
moon> find . -name myfile -print
./dir17/myfile
moon> vi $(find . -name myfile -print)
moon> date
Fri May 8 13:42:19 IDT 1998
moon> date | awk '{print $4}'
13:42:26
moon> echo "The time is $(date | awk '{print $4}')"
The time is 13:42:32
```

- צורה זו של המרה שימושית במיוחד ב-Shell scripts, עבור סריקת רשימה ארוכה של קבצים או מחרוזות המתקבלות כפלט של פקודות שונות במערכת:

```
#!/bin/bash
while sleep 60
do
TIME=$(date | awk '{print $4}')
case $TIME in
12:00*)
Run some job for 12:00 ;;
```

```
13:00*)
Run some other job at 13:00 ;;
14:00*)
Run some other job at 14:00 ;;
15:00*)
Run some other job at 15:00 ;;
19*|20*|21*|22*)
echo "This is the time to stop! ;;
23*|24*|01*)
echo "Are you still here!?" ;;
02*)
echo "I am logging you out!" ;;
    for proc in $(ps h | awk '{print $1}')
    do
        kill -9 $process
    done
esac
done
```

- בסקריפט הבא ניצור את פקודה בשם zap, אשר תקבל מחרוזת תווים ותסיים את כל התהליכים ששמם תואם את המחרוזת. הפקודה הרגילה kill אינה תמיד נוחה משום שיש לתת לה את מספר התהליך.

```
#!/bin/sh
USAGE="Form: $0 process-name.
This command accepts one argument:
the process name or pid, and it terminates
all processes by this name or pid."
if test $# != 1
```

```

then
echo $USAGE
exit -1
fi
for p in $(ps h | grep $1 | awk '{print $1}')
do
kill -9 $p
done

```

- הדוגמא הבאה מתבססת על הפקודות `who`, `wc`, ומראה איך אפשר להשתמש בפקודות פשוטות וקטנות בכדי לבנות פקודות חדשות ומעניינות.

```

moon> who
root      tty1      May  6 19:14
root      tty0      May  7 11:14 (:0.0)
root      tty1      May  8 13:16 (:0.0)
user18    tty2      May  8 14:24 (moon:0.0)
user32    tty3      May  8 14:46 (moon)
user32    tty2      May  8 14:57 (moon)
user32    tty7      May  8 14:11 (:0.0)
dany      tty8      May  8 09:54 (:0.0)
moon> wc /etc/passwd
268  831 17207 /etc/passwd
# This is the count of lines, words, chars.
moon> who | wc -l
8
moon> echo "There are $(who | wc -l) users logged"
There are 8 users logged

```

- **המרת ביטויים אריתמטיים (arithmetic evaluation)**
ביטוי אריתמטי יומר על ידי הצורה `$(expr)` או `[$expr]` להלן רשימת הפעולות והיחסים המותרים לשימוש על פי סדר קדימות:

<code>(...)</code>	grouping
<code>~</code>	one's complement
<code>!</code>	negation
<code>* / %</code>	multiplication, division, remainder
<code>+ -</code>	addition, subtraction
<code><< >></code>	left and right shifts
<code>< > <= >=</code>	less, greater, less or equal, ...
<code>== != =~ !~</code>	equal, not equal, matches, no match
<code>&</code>	logical and
<code>^</code>	logical xor
<code> </code>	logical or
<code>&&</code>	conditional and
<code> </code>	conditional or

- אריתמטיקה תתבצע על מספרים שלמים בלבד (כולל שליליים).
ברירת המחדל היא מספרים עשרוניים (בסיס 10) אך ניתן לציין בסיס על ידי הצורה `.base#number`.
- המרת פרמטרים והמרת פקודות ביניים ימשיכו להתבצע גם בתוך צורה זו.
- **דוגמאות:**

```
moon> a=3 b=4
moon> echo [$a*$b]
```

```
12
moon> echo ${a*b}
12
moon> echo $[(a+7)*(b+6)]
100
```

- הסקריפט הבא מקבל מהמשתמש רשימת קבצים (גם עם סימנים מיוחדים) ומחזיר את מספר התווים הכולל המופיעים בהם. לשם כך נדרש חישוב אריתמטי:

```
#!/bin/bash
FileList="$@"
total=0
for file in $FileList
do
    if ! test -e $file
    then
        echo "$file: file does not exist!"
        exit
    fi
    if test -f $file
    then
        total=$((total+$(wc -c $file | awk '{print $1}'))
    fi
done
echo "Total size=$total"
```

- **תרגיל:** הקלד את הסקריפטים השונים שניתנו עד עכשיו לחשבונך האישי לתוך ספרייה בשם bin (וודא כי ספרייה זו כלולה במשתנה הסביבתי PATH). בדוק אם הם עובדים, ונסה לערוך בהם שיפורים משלך (חלק מהם סובלים מבעיות שונות).

8.11 קיצורים (aliases)

- **קיצור** (alias) הוא שם נרדף לפקודה שמשתמשים בה הרבה. קיצורים אפשר ליצור באמצעות הפקודה alias אשר כבר נדונה. קיצורים שאנו מעוניינים שיהיו קבועים צריכים להופיע באחד מקובצי האיתחול .bashrc או .bash_profile.

Syntax:

```
alias [name=[str]]  
unalias [-a] [name1 name2 ...]
```

Examples:

```
moon> alias md=mkdir  
moon> alias md  
alias md='mkdir'  
moon> alias  
alias copy='cp'  
alias md='mkdir'  
alias grn='grep -n'  
alias which='type -path'  
moon> unalias md grn which  
moon> unalias -a
```

8.12 בקרת משימות (Job Control)

- כזכור, ניתן לטפל בתהליכים (בצורה חלקית) על ידי שימוש בפקודה ps. אך פקודה זו אינה מספיקה וגם אינה נוחה משום שתהליכים מיוצגים בה על ידי מספרים גדולים ולא נוחים.

- מעבד הפקודות bash (כמו עוד מעבדי פקודות חדשים) מספק לנו סביבה נוחה יותר לניהול משימות. כל תהליך שנוצר על ידי המשתמש נקרא job (במקום process), ומקבל מספר סידורי קטן.

● הפקודה jobs

מציגה את רשימת כל המשימות שרצות כרגע

```
moon> jobs
[1]    Running    sort bigfile > file2 &
[2]    Stopped    grep foo > bar2 &
[3]    Running    gcc hobo.c -o hop &
[4] -  Stopped    xboard &
[5] +  Stopped    make install &
```

הפקודה jobs עצמה רצה בחזית, ולכן אינה מופיעה ברשימה. הסימן + מסמן את המשימה המושהית האחרונה, - היא המשימה הקודמת.

- הפקודה bg משחררת משימה מושהית (stopped) להמשך עבודה ברקע. ללא ארגומנטים, פעולה זו תבצע על המשימה האחרונה שהושהתה. אחרת, יש לציין ארגומנט בצורה %n, כאשר n מציין מספר משימה כמו בטבלה הנ"ל.

- הפקודה fg מחזירה משימה מהרקע (background) לחזית (foreground).

```
moon> bg %2
```

```
moon> jobs
[1]   Running   sort bigfile > file2 &
[2]   Running   grep foo > bar2 &
[3]   Running   gcc hobo.c -o hop &
[4] - Stopped   xboard &
[5] + Stopped   make install &
moon> fg %5
```

History and fc commands 8.13

- בכל פעם שהמשתמש מפעיל פקודה, הפקודה נכנסת לרשימה שהמעבד שומר בדרך כלל בקובץ `..bash_history`.
- המשתנה הסביבתי `HISTSIZE` מציין את המספר המקסימלי של פקודות שהמעבד יישמור ברשימת ההיסטוריה (ברירת המחדל היא 1000).
- המשתנה הסביבתי `HISTFILE` מציין את הקובץ שבו תשמר ההיסטוריה. ברירת המחדל היא `~/bash_history`.
- הפקודה הפנימית `history` תציג את רשימת כל הפקודות השמורות בקובץ זה עד לרגע הנתון.

```
moon> history
124  ls -l /etc/X11 &
126  man 5 Xdefaults
127  startx
```



```

128  shutdown -h now
129  whoami
130  id
131  vi $(find . -name myfile -print)
132  kill $(ps | grep xboard | awk '{print $1}')
. . . . .
1122 vi .bash_history
1123 echo $HISTSIZE
moon>!132

```

בשורה האחרונה הפעלנו שוב את הפקודה שמספרה 132. מומלץ להשתמש בקיצור: `alias h=history`.

- לכל פקודה ברשימת ההיסטוריה מתאים מספר סידורי. אפשר להשתמש בפקודה `fc` בכדי להפעיל פקודה או קבוצת פקודות ישנות. בשלב הראשון רשימת הפקודות תוצג במעבד תמלילים (`vi` או `emacs` בהתאם למשתנה `FCEDIT` או `EDITOR`). לאחר שהמשתמש יערוך את רשימת הפקודות ויבצע שמירה, רשימת הפקודות תתבצע. צירוף שימושי במיוחד הוא `fc -s`.

```

moon> alias r='fc -s'
moon> r ps
# Run the last comand that starts with ps.
# Read the manual: help fc

```

8.14 פקודות פנימיות שכיחות

- `exec cmd`

פקודה זו גורמת למעבד להחליף את עצמו עם הפקודה `cmd`.

כלומר, המעבד מסיים את עצמו ומעביר את השליטה לפקודה `cmd`. לאחר סיום הפקודה `cmd` תתבצע יציאה מהמערכת (אין מעבד!).

- `file`
`source file`
 הפעלת ישירה של `file` (שהוא כמובן סקריפט). סקריפטים מופעלים בדרך כלל בצורה לא ישירה על ידי תת-מעבד. אך בצורה הנ"ל הם מופעלים ישירות בתוך המעבד המקורי, ולכן הם עשויים לשנות את המשתנים הסביבתיים, הספרייה הנוכחית, ומצב הקבצים הפתוחים (זה רצוי לפעמים).

- `exit [n]`
 יציאה מהמעבד. אם הארגומנט `n` קיים אז קוד היציאה הוא `n`. אחרת קוד היציאה הוא זה של הפקודה האחרונה שהתבצעה על ידי המעבד.

- `return [n]`
 יציאה מתוך פונקציה עם קוד יציאה `n`. ללא הארגומנט `n`, קוד היציאה הוא זה של הפקודה האחרונה שהתבצעה בתוך הפונקציה.

- `break [n]`
 יציאה מתוך `n` לולאות `while`, `until`, `for`. ללא הארגומנט `n` יציאה מתוך הלולאה הפנימית ביותר (כלומר $n = 1$).

- `continue [n]`

יציאה מתוך $n - 1$ לולאות `while`, `until`, `for`, והמשך הלולאה שכוללת אותן. ללא הארגומנט `n` יציאה חזרה ללולאה הנוכחית.

- `: [text]`

פקודה ריקה. למרות זאת שימושית לאיתחול משתנים או פתיחת קבצים.

- `trap [[cmdtext] signal1 signal2 ...]`

לכידת סיגנלים שנשלחים בזמן ביצוע הסקריפט וביצוע רשימת הפקודות `cmdtext` בתגובה לסיגנלים אלה. דוגמאות:

```
trap 'rm -f $tempfile; exit' 0 1 2 15
trap 'echo "cleaning up"
      rm tmp*
      exit 2' 0 1 2 15
```

- ללא ארגומנטים, הפקודה `trap` תחזיר את רשימת כל הסיגנלים כשלצידם הפעולה שתתבצע אם הם ייתקבלו:

```
moon> trap 'rm -f $tempfile; exit' 0 1 2 15
. . . . (later)
moon> trap
trap -- 'rm -f $tempfile; exit' EXIT
trap -- 'rm -f $tempfile; exit' SIGHUP
trap -- 'rm -f $tempfile; exit' SIGINT
trap -- 'rm -f $tempfile; exit' SIGTERM
```

- סיגנל מספר 0 נגרם בתוצאה מיציאה מהמעבד.
- אם `cmdtext` הוא המחרוזת הריקה אז הסקריפט יתעלם מכל הסיגנלים שמופיעים בפקודה. יוצא דופן הוא סיגנל SIGKILL שמספרו 9, אשר לא ניתן להתעלם ממנו.

```
trap '' 1 2 15
=> ignore signals 1, 2, and 15
```

- `unset [-vf] [name1 name2 ...]`
מחיקת משתנים (-v) או פונקציות (-f). לא ניתן למחוק את המשתנים EUID, UID, PS2, PS1, PPID, IFS, PATH.

- `shift [n]`
מיספור חדש של הארגומנטים: ארגומנט $n + 1$ הופך להיות 1, ארגומנט $n + 2$ הופך להיות 2, וכן הלאה. ארגומנטים 1 עד n לא קיימים יותר. ללא הארגומנט n , ברירת המחדל היא 1. פעם היתה זו הדרך היחידה לגשת לארגומנטים שמספרם עלה על 9.

- `set [-abefhkmnptuvxldCHP] [-o option] [arg1 arg2..]`
זוהי פקודה מורכבת מאוד בעלת מספר גדול של אופציות. לא נוכל לדון כאן בכולם. נתמקד בכמה דוגמאות חשובות בלבד. ניתן לקבל פירוט מלא באמצעות הפקודה "help set".

```
set -a var1 var2 . . . varn
Mark variables which are modified
or created for export.

set -n
Read commands but do not execute them.
```

```
set -o option-name
```

Set the boolean variable corresponding to option-name:

allexport same as -a

braceexpand the shell will perform brace expansion

emacs use an emacs-style line editing interface

ignoreeof the shell will not exit upon reading EOF

noclobber disallow redirection to existing files

noexec same as -n

posix change the behavior of bash where the default operation differs from the

1003.2 standard to match the standard

emacs Use an emacs style command line editing

vi Use a vi style command line editing

```
set -v
```

Print shell input lines as they are read.

```
set -x
```

Print commands and their arguments as they are executed.

```
set +o option-name
```

Causes these flags to be turned off.

- שימוש חשוב ונפוץ במיוחד בפקודה `set` הוא עבור יצירת פרמטרי מקום:

```
set foo bar hobo poo
```

```
=> $1=foo
```

```
    $2=bar
```

```
    $3=hobo
```

```
    $4=poo
```

```
moon> date
Mon May 18 13:11:57 IDT 1998
moon> set $(date) ; echo $4
13:11:57
```

ניתן לבנות פונקציות וסקריפטים כגון:

```
Time() {
    set $(date)
    echo $4
}
```

- ללא שום ארגומנטים, הפקודה `set` תציג את כל המשתנים הסביבתיים וערכיהם.

- `wait [n]`

חכה לסיום תהליך מספר `n` והחזר קוד יציאה.

- `local [name[=value]]`

הגדרת משתנה לוקלי בתוך פונקציה. ללא ארגומנטים מקבלים את רשימת המשתנים הלוקליים בפונקציה. ניתן להשתמש בפקודה זו רק בתוך פונקציות.

- `hash [-r] [name]`

בכל פעם שהמשתמש מפעיל פקודה, המעבד מכניס את המסלול המלא שלה לטבלת `hash`, בכדי שבפעם הבאה שהפקודה תופעל זמן חיפושה יתקצר. הפקודה `hash` מציגה את הטבלה הזו על

המסך. האופציה -r מוחקת את הטבלה. אפשר לבקש גם לחדש מסלול של פקודה מסוימת .name

```
moon> hash
hits command
  2 /usr/bin/less
  1 /usr/bin/man
  5 /bin/vi
  4 /usr/bin/who
  3 /bin/ls
moon> mv /bin/vi /usr/bin
moon> hash vi
moon> hash
  2 /usr/bin/less
  1 /usr/bin/man
  0 /usr/bin/vi
  4 /usr/bin/who
  3 /bin/ls
moon> hash -r
moon> hash
No commands in hash table
```

8.15 משתנים סביבתיים בשימוש המעבד

- רשימת המשתנים הבאה אינה מלאה. מובאים רק משתנים סביבתיים חשובים במיוחד לשם הדגמה. לשם קבלת רשימה מלאה יש לקרוא את דף העזרה של .bash

● PPID

מספר התהליך שיצר את המעבד.

● RANDOM

משתנה זה מכיל מספר שלם אקראי. המספר משתנה מקריאה אחת לשניה.

● SECONDS

משתנה זה מכיל את הזמן בשניות שעבר מהרגע שבו המעבד החל לפעול.

● HOME

ספריית הבית של המשתמש.

● PATH

רשימת מסלולי החיפוש של פקודות חיצוניות.

● IFS

רשימת התווים המשמשים כמפרידים בין ארגומנטים (input field separators). ברירת המחדל היא space, tab, new-line. הפקודה read משתמשת במשתנה זה בכדי לקלוט נתונים מהמשתמש.

● MAIL

הקובץ בו שמור הדואר של המשתמש.

● MAILCHECK

בכל פרק זמן קבוע המעבד בודק אם המשתמש קיבל דואר (זה נעשה על ידי בדיקת זמן השינוי של הקובץ \$MAIL). זמן זה שמור במשתנה MAILCHECK וברירת המחדל היא כל 60 שניות.

● ENV

המסלול של הקובץ המופעל מייד בכל פעם שמעבד או תת־מעבד נכנס לפעולה. ברירת המחדל היא ~/.bashrc. תתי־מעבדים אינם מפעילים את קובץ האיתחול הסטנדרטי ~/.bash_history. לכן אם ברצוננו כי משתנים, קיצורים, או פונקציות מסוימות יעברו גם לתתי־מעבדים, יש לכלול אותם בקובץ \$ENV.

● SHELL

המסלול של תוכנית ההפעלה של המעבד. בדרך כלל /bin/bash.

● LINES

מספר השורות במסך.

● COLUMNS

מספר העמודות במסך.

● HISTFILE

הקובץ ההיסטוריה. ברירת המחדל היא ~/.bash_history.

HISTSIZE ●

המספר המקסימלי של פקודות שיישמרו בקובץ הנ"ל. ברירת המחדל היא 1000.

SHLVL ●

מספר שלם המציין את רמת ההפעלה של המעבד. בכניסה למערכת שווה 1. אם מפעילים מעבד נוסף מתוך המעבד הנוכחי המספר גדל באחד.

BASH_VERSION ●

מספר הגירסה של המעבד. הגירסה הנוכחית היא 1.14.7.

HOSTNAME ●

שם המחשב. למשל moon.

HOSTTYPE ●

סוג המעבד. למשל i386, i486, או i586.

LOGNAME ●

שם כניסה של בעל החשבון.

USER ●

זהה למשתנה LOGNAME.

UID ●

מספר המשתמש.

OSTYPE •

שם מערכת ההפעלה. כגון Linux, SunOs, Ultrix, וכדומה.

8.16 קובצי איתחול וסיום**• /etc/profile**

קובץ איתחול לכלל המערכת. בכל פעם שמעבד הפקודות bash מופעל, קובץ זה מופעל על ידו. קובץ זה מכיל את ההגדרות של המשתנים הסביבתיים הסטנדרטיים דוגמא לקובץ זה במחשב

:moon

```
# /etc/profile
# System wide environment and startup programs
# Functions and aliases go in /etc/bashrc
PATH="$PATH:/usr/X11R6/bin"
PS1="[\u@\h \W]\\$ "
ulimit -c 1000000
if [ 'id -gn' = 'id -un' -a 'id -u' -gt 14 ]; then
umask 002
else
umask 022
fi
USER='id -un'
LOGNAME=$USER
MAIL="/var/spool/mail/$USER"
HOSTNAME='/bin/hostname'
HISTSIZE=1000
HISTFILESIZE=1000
```

```
export PATH PS1 HOSTNAME HISTSIZE
export HISTFILESIZE USER LOGNAME MAIL
for i in /etc/profile.d/*.sh ; do
if [ -x $i ]; then
. $i
fi
done
```

• /etc/bashrc

קובץ איתחול שני לכלל המערכת. בניגוד לקובץ הראשון, קובץ זה אינו מופעל אוטומטית על ידי המעבד! הפעלת הקובץ מתבצעת בדרך כלל מתוך אחד מקובצי האיתחול של המשתמש (במידה והוא מעוניין בכך). קובץ זה מכיל בדרך כלל קיצורים ופונקציות לשימוש כלל המשתמשים. דוגמא לקובץ זה במחשב

```
:moon
```

```
# /etc/bashrc
# System wide functions and aliases
# Environment stuff goes in /etc/profile
# PS1="[ \u@\h \W]\\$ "
PS1="\u: \w> "
alias which="type -path"
alias dir='ls -CFs'
```

• ~/.bash_profile

קובץ איתחול פרטי לכל משתמש ומשתמש. לכל משתמש יש קובץ כזה בספריית הבית שלו (קובץ נסתר). בכל פעם שהמשתמש נכנס למערכת קובץ זה מופעל על ידי bash. קובץ

זה מכיל בדרך כלל הגדרות יותר ספציפיות של כמה משתנים הסביבתיים. דוגמא לקובץ מינימלי כזה במחשב moon:

```
# .bash_profile
# Get the aliases and functions
if [ -f ~/.bashrc ]; then
    . ~/.bashrc
fi
# User specific environment and startup programs
PATH=$PATH:$HOME/bin
ENV=$HOME/.bashrc
USERNAME="Rony Levy"
export USERNAME ENV PATH
```

זהו הקובץ הנמסר לכל משתמש עם פתיחת חשבוננו. המשתמש יוכל לערוך בו שינויים כרצונו ולהתאים אותו לטעמו האישי.

● `~/.bashrc`

קובץ איתחול פרטי לכל משתמש ומשתמש. לכל משתמש יש קובץ כזה בספריית הבית שלו (קובץ נסתר). זהו אינו קובץ איתחול רישמי של `bash`! כפי שרואים לעיל, קובץ זה מופעל מתוך הקובץ `.bash_profile`, ולכן אפשר לדלג עליו אם רוצים. הסידור המקובל הוא שקובץ זה יכיל בדרך כלל קיצורים ופונקציות שהמשתמש מעוניין שיעברו הלאה לכל המעבדים ותתי-המעבדים שהוא עשוי להפעיל במהלך עבודתו. אך אפשר לכלול בתוכו גם הגדרות של משתנים הסביבתיים. המשתנה הסביבתי `ENV` מצביע על קובץ זה. כזכור תתי-מעבדים

מפעילים את הקובץ \$ENV בלבד. דוגמא לקובץ מינימלי כזה
במחשב moon:

```
# .bashrc
# User specific aliases and functions
# Source global definitions
if [ -f /etc/bashrc ]; then
. /etc/bashrc
fi
```

זהו הקובץ הנמסר לכל משתמש עם פתיחת חשבוננו. המשתמש
יוכל לערוך בו שינויים כרצונו ולהתאים אותו לטעמו האישי.

- `/.bash_logout`

אם קובץ זה קיים, הוא יופעל לפני יציאה מהמעבד.

- **דוגמא לקובץ `.bash_profile` יותר מפותח:**

```
# Hello, my name is chet!
# and this is my .bash_profile file:
HOME=/usr/homes/chet
MAIL=/usr/homes/chet/mbox
MAILCHECK=30
HISTFILE=/usr/homes/chet/.history
HOST=$(hostname)
PATH1=$HOME/bin.$HOSTTYPE:/usr/local/bin/gnu:
PATH2=/usr/local/bin:/usr/ucb:/bin:/usr/bin/X11:.
PATH3=/usr/andrew/bin:/usr/bin:/usr/ibm:
PATH4=/usr/local/bin/mh:/usr/new/bin:
PATH=$PATH1:$PATH2:$PATH3:$PATH4
EDITOR=/usr/homes/chet/bin.$HOSTTYPE/ce
```

```
VISUAL=/usr/homes/chet/bin.$HOSTTYPE/ce
FCEDIT=/usr/homes/chet/bin.$HOSTTYPE/ce
if [ "$HOSTTYPE" = "ibm032" ] ; then
    stty erase ^H
fi
PAGER=/usr/ucb/more
NNTPSERVER=kiwi
#
# Bogus 1003.2 variables.
# This should really be in /etc/profile
LOGNAME=${USER-$(whoami)}
TZ=EST5EDT
export HOME ENV VISUAL EDITOR MAIL SHELL PATH TERM
export PAGER LESS TERMCAP HISTSIZE HISTFILE
export MAIL MAILCHECK HOST HOSTNAME
export NNTPSERVER NS LOGNAME TZ
PS1="${HOST}$ "
PS2='> '
export PS1 PS2
umask 022
if [ -f /unix ] ; then
    stty intr ^c
fi
if [ -f ~/.bashrc ] ; then
    . ~/.bashrc
fi
```

• דוגמה לקובץ .bashrc יותר מפותח:

```
if [ "$PS1" != "" ] ; then
if [ -f /unix ] ; then
```

```
alias ls='/bin/ls -CF'
alias ll='/bin/ls -lCF'
alias dir='/bin/ls -bCaF'
else
alias ls='/bin/ls -F'
alias ll='/bin/ls -lF'
alias dir='/bin/ls -baF'
fi
alias ss="ps aux"
alias mail=/usr/ucb/mail
alias dot='ls .[a-zA-Z0-9]*'
alias mroe=more
alias pwd='echo $PWD'
alias pdw='echo $PWD'
alias news="xterm -g 80x45 -e rn -e &"
alias back='cd $OLDPWD'
alias manroff="nroff /usr/lib/tmac/tmac.an.4.3"
alias laser="lpr -Palw2"
alias lw="lpr -Palw2"
alias c="clear"
alias m="more"
alias j="jobs"
if [ -z "$HOST" ] ; then
    export HOST='hostname'
fi
history_control=ignoredups
psgrep()
{
    ps -aux | grep $1 | grep -v grep
}
```



```
#
# This is a little like 'zap'
# from Kernighan and Pike
#
pskill()
{
    local pid
    pid=$(ps ax |grep $1 |grep -v grep |awk '{ print $1 }')
    echo -n "killing $1 (process $pid)..."
    kill -9 $pid
    echo "slaughtered."
}

term()
{
    TERM=$1
    export TERM
    tset
}

cd()
{
    builtin cd $*
    xtitle $HOST: $PWD
}

bold()
{
    tput smso
}

unbold()
{
    tput rmso
}
```

```
}
if [ -f /unix ] ; then
clear()
{
    tput clear
}
fi
rot13()
{
    if [ $# = 0 ] ; then
        tr "[a-m][n-z][A-M][N-Z]" "[n-z][a-m][N-Z][A-M]"
    else
        tr "[a-m][n-z][A-M][N-Z]" "[n-z][a-m][N-Z][A-M]" < $1
    fi
}
watch()
{
    if [ $# -ne 1 ] ; then
        tail -f nohup.out
    else
        tail -f $1
    fi
}
#
# Remote login passing all 8 bits
# (so meta key will work)
#
rl()
{
    rlogin $* -8
}
}
```

```
function setenv()
{
    if [ $# -ne 2 ] ; then
        echo "setenv: Too few arguments"
    else
        export $1="$2"
    fi
}
function chmog()
{
    if [ $# -ne 4 ] ; then
        echo "usage: chmog mode owner group file"
        return 1
    else
        chmod $1 $4
        chown $2 $4
        chgrp $3 $4
    fi
}
fi
#end of .bashrc
```

● דוגמא לקובץ `.bash_aliases`

בניגוד לשני הקבצים הקודמים, קובץ זה אינו מופעל אוטומטית על ידי `bash`. יש הפעיל אותו לכן מתוך אחד משני הקבצים באמצעות פקודת `source` למשל.

```
# Some useful aliases.
alias texclean='rm -f *.toc *.aux *.log *.cp
                *.fn *.tp *.vr *.pg *.ky'
```

```
alias clean='echo -n "Really clean this directory?";
  read yorn;
  if test "$yorn" = "y"; then
    rm -f *~ .*~ *.bak *.bak *.tmp *.tmp core a.out;
    echo "Cleaned.";
  else
    echo "Not cleaned.";
  fi'
alias h='history'
alias j="jobs -l"
alias l="ls -l "
alias ll="ls -l"
alias ls="ls -F"
alias term='set noglob; eval `tset -Q -s `
alias pu="pushd"
alias po="popd"
#
# Csh compatability:
#
alias unsetenv=unset
function setenv () {
  export $1="$2"
}
# Function which adds an alias to the current shell and to
# the ~/.bash_aliases file.
add-alias ()
{
  local name=$1 value="$2"
  echo alias $name=\'$value\' >> ~/.bash_aliases
  eval alias $name=\'$value\'
  alias $name
```

```
}
# "repeat" command. Like:
#
#     repeat 10 echo foo
repeat ()
{
    local count="$1" i;
    shift;
    for i in $(seq 1 "$count");
    do
        eval "$@";
    done
}
# Subfunction needed by 'repeat'.
seq ()
{
    local lower upper output;
    lower=$1 upper=$2;
    while [ $lower -le $upper ];
    do
        output="$output $lower";
        lower=$(( lower + 1 ));
    done;
    echo $output
}
```

• דוגמה לקובץ `.bash_functions`

קובץ זה אינו מופעל אוטומטית על ידי `bash`. יש הפעיל אותו לכן

מתוך אחד משני הקבצים `.bashrc` או `.bash_profile`, באמצעות פקודת `source` למשל.

```
existfile () { if test $1;then test -f $1;else false;fi }
textfile () { file ${1?} | grep text > /dev/null }
rmf () { if test $1;then rm -f $@;fi; }
mvf { if test $2;then mv -f $@;fi }
lines () {
    local file=$1 top=$2 nbr=$3
    head -n $top $file | tail -n $nbr
}
line () {
    local file=$1 top=$2
    filelines $file $top 1
}
userpids () { ps u | grep "^$1.*" | cut -b 8-15 }
setbool () { case $1 in
    y|Y|+) echo 1;;
    n|N|-) echo 0;;
    *) return 1;;esac;
}
function isuser { id $1 >/dev/null 2>&1 }
function users () {
    cd /home
    for D in $(subdirs)
    do
    if isuser $D
    then echo $D;fi
    done
}
}
```

```
pathls () { for F in $PWD/*;do echo $F;done }
chowngrp () {
    chown -R $1 $3
    chgrp -R $2 $3
    chmod -R u+rwX,g+r $@
}
stringeq () { test k$1 = k$2; }
prepend () {
    cat $1 $2 > /tmp/prepend
    rm $1;mv -f /tmp/prepend $2
}
uppercase () {
    tr abcdefghijklmnopqrstuvwxyz ABCDEFGHIJKLMNOPQRSTUVWXYZ
}
lowercase () {
    tr ABCDEFGHIJKLMNOPQRSTUVWXYZ abcdefghijklmnopqrstuvwxyz
}
beep () { echo -e "\007" >/dev/console }
# Directory manipulation functions
# from the book 'The Korn Shell'
unalias cd
# alias cd=_cd
# Menu-driven change directory command
function mcd
{
    dirs
    echo -n "Select by number or enter a name: "
    read
    cd $REPLY
}
}
```

פרק 9

ביטויים רגולריים

9.1 (Regular Expressions)

- ביטוי רגולרי הוא ביטוי המתאר קבוצה של מילים בעלי צורה משותפת.
 - אם word היא מילה המתוארת על ידי ביטוי רגולרי exp אז נומר כי המילה word תואמת את הביטוי הרגולרי exp.
 - מספר גדול של יישומים ושפות תיכנות (כגון awk, grep, tcl, emacs, vi, perl) מתבססות על ביטויים רגולריים, מה שמחייב את הבנת הנושא כבר בשלב מוקדם.
 - אין לבלבל בין ביטויים רגולריים ובין **ביטויים גלובליים** אשר בהם משתמשים המעבדים השונים.
 - קיימים שני סוגים של ביטויים רגולריים:
- ביטויים רגולריים בסיסיים - Basic Regular Expressions

(BRE)

Extended Regular Expressions (ERE) - ביטויים רגולריים מורחבים -

● בדרך כלל כל תו המופיע בתוך ביטוי רגולרי מתאר את עצמו בלבד. למשל הביטוי הרגולרי `braude` מתאר את המילה `braude` בלבד.

● יוצאים מהכלל הם התאים המיוחדים הבאים:

- \ * [

- התו `^` הוא תו מיוחד בתנאי שהוא מופיע בתחילת הביטוי.

- התו `$` הוא תו מיוחד בתנאי שהוא מופיע בסוף הביטוי.

-

● אם רוצים להשתמש בתו מיוחד כתו רגיל יש לצטט אותו על ידי קו נטוי (`\`).

● לדוגמא, הביטוי הרגולרי `'price is $9\.95'` מתאר את המחרוזת `'price is $9.95'` בלבד.

● המשמעות של התווים המיוחדים:

\ מצטט את התו הבא מייד אחריו, אם תו זה הוא תו מיוחד.
אם התו הבא אינו תו מיוחד התוצאה אינה מוגדרת.

. תואם כל תו.

* תואם אפס או יותר חזרות של התו הקודם או של הביטוי הקודם. למשל כל מילה (גם ריקה!) תואמת את הביטוי הרגולרי `*..` שים לב להבדל הגדול לשימוש בתו זה בביטויים גלובליים! למשל הביטוי `*.txt` אינו ביטוי רגולרי! זהו ביטוי גלובלי בשימוש המעבד `bash` למשל. ביטוי רגולרי שקול הוא `.*\.`

^ אם תו זה מופיע בתחילת ביטוי רגולרי אז הוא תואם התחלת שורה. בכל מקום אחר הוא תואם את עצמו בלבד.

\$ אם תו זה מופיע בסוף ביטוי רגולרי אז הוא תואם סוף שורה. בכל מקום אחר הוא תואם את עצמו בלבד.

[set]

המונח `set` מציין קבוצת תווים אשר ניתן להגדיר בכמה דרכים שונות. הביטוי תואם כל תו המשתייך לקבוצה. סוג זה של ביטוי רגולרי נקרא **ביטוי מסגרת**. למשל הביטוי `[mogly]` תואם כל אחד מהתווים `y, l, g, o, m`. צורה שניה: למשל הביטוי `[c-k]` תואם כל אחד מהתווים בין `c` ל-`k`.

הביטוי `[0-9]` תואם כל אחד מהספרות בין `0` ל-`9`. הביטוי `[a-z0-9]` תואם כל אחד מאותיות האנגליות הקטנות או הספרות בין `0` ל-`9`.

צורה שלישית: יישומים המצייתים לתקן POSIX כוללים גם צורה חדשה זו. ביטוי שצורתו `[:class:]` תואם כל אחד מהתווים המשתייכים לקבוצה ששמה `class`. הקבוצה `class` יכולה להיות כל אחת מהקבוצות הבאות:

<code>upper</code>	<code>A-Z</code>
<code>lower</code>	<code>a-z</code>
<code>alpha</code>	<code>a-z, A-Z</code>

digit	0-9
xdigit	012345678abcdefABCDEF
space	space, tab, newline, cr, ff, vtab
blank	space, tab
cntrl	non-printable chars (ASCII 0-31, 127)
punct	all except digit, alpha, xdigit !"#\$%&'()*+,-./:;<?>=@[\]^_`{}`
graph	graphic characters: alpha, digit, punct
print	printable characters: graphic, space

- הביטוי הרגולרי `[^set]` מתאר את הקבוצה המשלימה. למשל הביטוי הרגולרי `[^0-9]` מתאר את כל התווים שאינם ספרות.
- כנ"ל לגבי `[^[:class:]]`. למשל הביטוי הרגולרי `[^[:punct:]]` מתאר את כל התווים שאינם תווי פיסוק.
- שאלה: האם המחרוזת `axybbcc` התואמת את הביטוי הרגולרי `?axccybb^a.*b.c`
- הסוגר `]`, אם אינו מסיים קבוצה בצורה חוקית תואם את עצמו בלבד.
- **דוגמאות:**
 - `dany*` מתאר את המילים: `dan, dany, danyy, danyyy, ...`
 - `[[:digit:]]a-fA-f` מתאר כל תו שהוא ספרה או תו `a-f` או `A-F`.
 - `[[:digit:]]a-fA-f`* מתאר כל המחרוזות המורכבות מהתווים הנ"ל (מספרים הקסדצימליים שלמים).

- * [a-zA-Z0-9] [a-zA-Z] מתאר כל המחרוזות שהם שמות של משתנים חוקיים בשפת C.

- * [0-9] [0-9] [1-9] מתאר כל המספרים העשרוניים הגדולים מ-9.

- * [01] מתאר את כל המחרוזות הבינאריות.

9.2 תתי-ביטויים (Subexpressions)

- כל ביטוי רגולרי שנמצא בין הסוגר (\ לסוגר) \ נקרא תתי-ביטוי.
- ביטוי רגולרי יוכל להיות מורכב מכמה תתי-ביטויים. במסגרת הביטוי ניתן להתייחס לכל אחד מתתי-הביטויים באמצעות הייחוס n \ כאשר n הוא מספר בין 0 ל-9 המציין את מקומו של תתי-הביטוי בתוך הביטוי כולו.
- למשל הביטוי הרגולרי $\backslash([a-z]\backslash)^*=\backslash 1$ מתאר את כל המחרוזות מהצורה $word=word$, כאשר $word$ היא כל מילה אנגלית המורכבת מאותיות קטנות.
- הביטוי הרגולרי $\backslash(/[a-z][a-z]^+\backslash)^+$ מתאר את כל המחרוזות מהצורה $/word1/word2/.../wordn$, כאשר $word1, \dots, wordn$ הן מילים אנגליות המורכבת מאותיות קטנות בעלי שתי אותיות לפחות.

- ביטויים אינטרבליים: ניתן להוסיף לכל תו או ביטוי רגולרי המתאר קבוצת תווים את אחד מהסיומות הבאות:

$\{m\}$ $\{m,\}$ $\{m,n\}$

כאשר m, n מציינים מספרים טבעיים בין 1 ל-256. התוצאה היא:

- במקרה הראשון: ביטוי רגולרי המתאר מחרוזות באורך m המורכבות מהתו שבא לפני הסיומת.
- במקרה השני: ביטוי רגולרי המתאר מחרוזות באורך m ומעלה המורכבות מהתו שבא לפני הסיומת.
- במקרה השלישי: ביטוי רגולרי המתאר מחרוזות באורך m עד n המורכבות מהתו שבא לפני הסיומת.

● דוגמאות:

- הביטוי $\{17\} [0-9]$ מתאר את כל המספרים העשרוניים בעלי בדיוק 17 ספרות.
- הביטוי $\{5,\} [0-9a-zA-F]$ מתאר את כל המספרים ההקסדצימליים בעלי לפחות 5 ספרות.
- הביטוי $\{5,12\} [a-zA-Z]$ מתאר את כל המחרוזות האנגליות בעלי 5 עד 12 תווים. 5 ספרות.

9.3 ביטויים רגולריים מורחבים

- ההגדרה של **ביטויים רגולרי מורחבים** כוללת את כל הכללים של ביטויים רגולריים בסיסיים יחד עם הכללים הנוספים הבאים:
 - לרשימת התווים המיוחדים מתווספים התווים הבאים:
! + ? | () " !
 - (התווים) ו- משמשים לסמן תתי-ביטויים כמו קודם.
 - +
- תואם אחד או יותר חזרות של התו הקודם או של הביטוי הקודם. למשל, אם exp הוא ביטוי רגולרי מורחב אז $(exp)^+$ הוא ביטוי רגולרי מורחב המתאר את אוסף כל המילים שהם שירשור של מילה אחת או יותר שתואמות לביטוי exp . למשל המילים aaa , ab , abb , $abab$, כולן תואמות את הביטוי הרגולרי המורחב $(ab^*)^+$.
- ?
 - תואם אפס או פעם אחת את התו הקודם או את הביטוי הקודם. למשל, הביטוי הרגולרי המורחב $abc?.txt$ מתאר שני מילים בלבד: $ab.txt$, $abc.txt$. שים לב להבדל בשימוש בתו זה בביטויים גלובליים! למשל, במסגרת המעבד `bash`, לביטוי הגלובלי $abc?.txt$ יש משמעות אחרת לגמרי.
 - |

אם $exp1$, $exp2$ הם שני ביטויים רגולריים מורחבים אז $exp1|exp2$ הוא ביטוי רגולרי מורחב המתאר את אוסף כל המילים התואמות את $exp1$ או את $exp2$.

● **דוגמאות:**

- $(cat|dog)(foo|bar)$

מתאר ארבעה מילים בדיוק: $dog-$, $dogfoo$, $catbar$, $catfoo$.
 $.bar$

- $.*\.(zip|exe|obj)$

מתאר את שמות כל הקבצים עם סיומת $.zip$, $.exe$, או $.obj$.

- $[01]^+$ מתאר את כל המחרוזות הבינאריות.

- $0|([1-9][0-9]^*)$

מתאר את כל המספרים העשרוניים השלמים האי-שליליים (שים לב שהמספר 003901, למשל, אינו תואם את הביטוי!).

- $0|(\+|-)?([1-9][0-9]^*)$

מתאר את כל המספרים העשרוניים השלמים (כולל שליליים!).

- $(0(\.[0-9]^+)?)|([1-9]+(\.[0-9]^+)?)$

מתאר את כל המספרים הממשיים האי-שליליים.

9.4 חיפוש והחלפת ביטויים רגולריים

- יישומים רבים ושפות תיכנות רבות כוללות בתוכן פקודות סריקת טקסט לאיתור מחרוזות תווים התואמות ביטוי רגולרי נתון (או רשימת ביטויים רגולריים). למשל: `grep`, `egrep`, `fgrep`, `vi`, `emacs`, `sed`, `awk`, `Perl`, `Tcl`, ועוד.

- דפדפני אינטרנט כגון Netscape או Explorer כוללים בתוכם סורקים של קובצי HTML לאיתור כתובות URL וכדומה.

A regular expression for searching http URL's:

```
<A *href=http://.*>
```

- בנוסף לאיתור מחרוזות, מרבית היישומים הנ"ל כוללים פקודות לשם החלפת המחרוזות שאותרו במחרוזות חדשות.

- פקודות כאלה יוצרות קושי מסוים משום שברוב המקרים קיים מספר גדול של אפשרויות שבה ביטוי רגולרי עשוי לתאום תת-מחרוזת של טקסט נתון:

```
reg exp:  a.*a
text:     "don't say abra cada bra not"
There are 9 matching substrings:
"ay a",   "ay abra",   "ay abra ca",
"ay abra cada",   "ay abra cada bra",
"abra cada bra",   "abra cada",
"abra ca",   "abra"
```


- ברוב היישומים, פקודת החיפוש תחזיר תת-מחרוזת אחת בלבד מבין כל תת-המחרוזות התואמות, ולכן, ברוב המקרים היא צריכה להתעלם מתת-מחרוזות תואמות נוספות.
- ההסכם המקובל בכל היישומים הוא: ברירת המחדל של פקודת החיפוש תהיה להחזיר תמיד את תת-המחרוזת שהיא קודם כל ראשונה ומבין כל הראשונות את הארוכה ביותר.

9.5 ביטויים רגולריים ב-Awk

- הביטויים הרגולריים בשפת התיכנות Awk הם ביטויים רגולריים מורחבים כפי שהגדרנו קודם. השפה כוללת בתוכה כמה פונקציות לטיפול בביטויים רגולריים:

`match(s, r)`

returns the position in `s` where the regular expression `r` occurs, or 0 if `r` is not present, and sets the values of `RSTART` and `RLENGTH`.
`RLENGTH`=length of matching substring
`RSTART` =start of matching substring

`split(s, a [, r])`

splits the string `s` into the array `a` on the regular expression `r`, and returns the number of fields.
 If `r` is omitted, `FS` is used instead.

`sub(r, s [, t])`

Just like `gsub()`, but only the first matching substring is replaced.

`gsub(r, s [, t])`

For each substring matching the reg expr `r` in the string `t`, substitute the string `s`, and return the number of substitutions.

If `t` is not supplied, use `$0`.

An `&` in the replacement text is replaced with the text that was actually matched.

(Use `\\&&` to get a literal `&`)

`gensub(r, s, h, [, t])`

Search the target string `t` for the reg expr `r`.

If `h` is a string beginning with `g` or `G` then replace all matches of `r` with `s`.

Otherwise, `h` is a number indicating which match of `r` to replace.

If no `t` is supplied, `$0` is used instead.

● **דוגמא 1:** הסקריפט הבא ממיין שורות של קובץ על פי האורך שלהן:

```
#!/bin/bash
### lensort - sort by line length
### Usage: lensort [files]
# print each line's length, a TAB and
# then the actual line
awk '{ printf "%d\t%s\n", length($0), $0 }' |
# Sort the lines numerically
```

```
sort -n -k1 |
# Remove the length and the TAB
# and print each line
awk '{sub(/^ [0-9] [0-9]*\t/, ""); print $0}'
```

• דוגמא 2:

```
#!/bin/bash
pattern="$1"; shift
awk '# getmatch -- print string that matches line
{ # extract string matching pattern using
  # starting position and length of string in $0
  if (match($0, pattern))
    print substr($0, RSTART, RLENGTH)
}' pattern="$pattern" $*
```

• אם נפעיל את הפקודה: `getmatch a.*a myfile`

כאשר `myfile` הוא הקובץ הבא:

```
This is the time for all good
people to say abracadabra
don't tell abra cada bra
do say aBRA CADA BRA
```

יתקבל הפלט הבא:

```
moon> getmatch a.*a myfile
ay abracadabra
abra cada bra
ay a
```

● דוגמא 3:

```

#!/bin/bash
# showmatch:
# mark string that matches pattern
# find starting position and length of matching string
# in input print the line where the match was found
pattern="$1"; shift
awk '{
  if (match($0, pattern)) {
    print
    # create a string of ^ the length of the match.
    patmatch = ""
    for (k = 1; k <= RLENGTH; k++)
      patmatch = patmatch "^"
    # print a blank column followed by string of ^
    printf("%RSTART-1"s" "%-s\n", "", patmatch)
  }
}' pattern="$pattern" $*

```

● אם נפעיל את הפקודה: `showmatch a.*a myfile`

כאשר `myfile` הוא הקובץ הבא:

```

This is the time for all good
people to say abracadabra
don't tell abra cada bra
do say aBRA CADA BRA

```

יתקבל הפלט הבא:

```

moon> showmatch a.*a myfile
people to say abracadabra

```

```
          ~~~~~  
don't tell abra cada bra  
          ~~~~~  
do say aBRA CADA BRA  
      ~~~~
```

פרק 10

פיתרון תרגילים

להלן קובץ של תרגילים פתורים אשר ניתנו בקורסי מערכות הפעלה שונים במהלך שנות ה-90. בשנים ההם, מרבית הסקריפטים נכתבו בשפות סקריפט מסורתיות כמו `sh`, `bash`, `csh`. גם כיום קיים הכרח וצורך להכיר את שפות הסקריפט המסורתיות מאחר וקיימת כמות עצומה של קוד הכתובה בהם הנמצאת בשימוש יומיומי וקריטי בכל הווריאנטים השונים של מערכות ה-Unix למיניהם, וחברות מרכזיות בתחומי המיחשוב והאלקטרוניקה עדיין תלויים באופן קריטי בסקריפטים אלה.

למרות זאת, כיום שפות הסקריפט המודרניות (`Python`, `Tcl`, `Perl`), מאפשרות לנו לפתור את מרבית הבעיות המוצגות כאן באופן נוח ופשוט בהרבה, וקיים מעבר הולך וגובר משפות הסקריפט המסורתיות לחדשות במרבית התחומים. ההמלצה לתלמיד היא לנסות לפתור את הבעיות המובאות כאן בשתי הדרכים המוצעות: `BASH`, `Python`, כאשר הדגש הוא יותר על

שפת התיכנות Python. על כל פנים, התלמיד נדרש גם להכיר את שפת הסקריפט BASH, אם כי ברמה בסיסת מאוד.

תרגיל 1:

כתוב פונקציה, סקריפט, או pipeline, המקבל שם של ספרייה (יחסית או אבסולוטית), ופולט את שם הקובץ הגדול ביותר ששייך לה. ללא ארגומנט פעולה זו תתבצע על הספרייה הנוכחית.

פיתרון:

```
#!/bin/bash
#file name: biggest
# The script is based on "ls -S"
# which sorts by file size, with
# the biggest file first
case $# in
0) big=$(ls -S | head -1)
   echo "The biggest file is: $big" ;;
1) dir=$1
   if
       ! test -d $dir
   then
       echo "$dir: directory does not exist!"
       exit -1
   fi
   big=$(ls -S $dir | head -1)
   echo "The biggest file is: $big" ;;
*) echo "Wrong number of arguments!" ;;
esac
```

תרגיל 2:

כתוב פונקציה, סקריפט, או pipeline, המקבל שם של ספרייה (יחסית או אבסולוטית), ופולט את שם הקובץ הישן ביותר ששייך לה (mtime). ללא ארגומנט פעולה זו תבצע על הספרייה הנוכחית.

פיתרון:

```
#!/bin/bash
#file name: oldest
# The script is based on "ls -t"
# that sorts file according to
# modification time (newest file first!)
case $# in
0) old=$(ls -t | tail -1)
   echo "The oldest file is: $old" ;;
1) dir=$1
   if
       ! test -d $dir
   then
       echo "$dir: directory does not exist!"
       exit -1
   fi
   old=$(ls -t $dir | tail -1)
   echo "The oldest file is: $old" ;;
*) echo "Wrong number of arguments!" ;;
esac
```


תרגיל 3:

כתוב סקריפט בשם nopts המקבל מספר כלשהו של ארגומנטים ומציג את מספר הארגומנטים, הפרמטרים, והאופציות:

```
host> nopts aaa -bb -c ddd eeee -f g
arguments: 7
options: 3
parameters: 4
```

פיתרון:

```
#!/bin/bash
#file name: nopts
opts=0
params=0
for x in $@
do
case $x in
-*) opts=${opts+1} ;;
*) params=${params+1} ;;
esac
done
echo "Args=$#"
echo "Options=$opts"
echo "Parameters=$params"
```

תרגיל 4:

כתוב סקריפט שאינו מקבל ארגומנטים, אך מבקש מהמשתמש שנה וחודש (בצורה מילולית או מספרית), ומציג את לוח השנה המתאים. בשלב הראשון תוכל להתבסס על הפקודה cal של Unix. הפיתרון בשפת Python צריך להתבסס על חבילות Python בלבד.

פיתרון:

```
#!/bin/sh
#file name: Cal
echo -n "Enter month:"
read m
echo -n "Enter year:"
read y
case $m in
[jJ]an*)      m=1 ;;
[fF]eb*)      m=2 ;;
[mM]ar*)      m=3 ;;
[aA]pr*)      m=4 ;;
[mM]ay*)      m=5 ;;
[jJ]un*)      m=6 ;;
[jJ]ul*)      m=7 ;;
[aA]ug*)      m=8 ;;
[sS]ep*)      m=9 ;;
[oO]ct*)      m=10 ;;
[nN]ov*)      m=11 ;;
[dD]ec*)      m=12 ;;
```

```
[1-9]|10|11|12)      ;;
# A numeric month was given.
# No need to convert.
*)      echo "Error: llegal month value!"
        exit 1 ;;
esac
if
    test "[0<y && y<10000] != 1
then
    echo "$y: illegal year value!"
    exit 2
fi
/usr/bin/cal $m $y    # Call the system cal command.
```

תרגיל 5:

כתוב סקריפט בשם clock אשר הפעלתו תציג את השעה באופן הבא:

```
The time is HH o'clock MM minuts and SS seconds
```

כאשר HH, MM, SS הם השעות, דקות ושניות בהתאמה. יש להעזר בפקודה date.

פיתרון:

```
#!/bin/sh
set $(date)
time=$4
IFS=:
set $time
HH=$1 ; MM=$2 ; SS=$3
echo "The time is $HH o'clock $MM minutes\
    and $SS seconds"
```

תרגיל 6:

כתוב פונקציה או סקריפט של bash בשם day אשר תקבל שלושה ארגומנטים: יום בחודש, חודש, ושנה (בצורה מספרית) ותפלוט את היום המתאים בשבוע בצורה מילולית:

```
host> day 25 10 1948
```

```
Monday
```

```
host> day 31 1 2001
```

```
Wednesday
```

הדרכה: התבסס על פקודות כגון cal, grep, awk.

פיתרון:

הפלט של הפקודה cal נראה כך

```
September 1998
Su Mo Tu We Th Fr Sa
      1  2  3  4  5
 6  7  8  9 10 11 12
13 14 15 16 17 18 19
20 21 22 23 24 25 26
27 28 29 30
```

יש להתעלם משתי השורות הראשונות. זה מתבצע באמצעות awk על ידי הדפסת שורות שמספרן 2 ומעלה. לאחר מכן נקלוט את השורה שבה מופיע היום המתאים.

```
#!/bin/sh
if
    test $# != 3
then
```

```
    echo "Wrong number of args"
    exit 1
fi
d=$1 ; m=$2 ; y=$3
if
    test "[1<=d && d<=31]" != 1
then
    echo "Illegal day value. Must be 1-31."
    exit 2
fi
if
    test "[1<=m && m<=12]" != 1
then
    echo "Illegal month value. Must be 1-12."
    exit 3
fi
if
    test "[1<=y && y<=9999]" != 1
then
    echo "Illegal month value. Must be 1-9999."
fi
line=$(cal $m $y | awk 'NR>2 {print $0}' | grep -w $d)
if
    test -z "$line"
then
    echo "There is no $d day in month $m year $y!"
    exit 4
fi
pos=$(echo "$line" | awk "{print index(\$0,$d)}")
weekday=$((pos/3 + 1))
```

```
case $weekday in
1) echo Sunday ;;
2) echo Monday ;;
3) echo Tuesday ;;
4) echo Wednesday ;;
5) echo Thursday ;;
6) echo Friday ;;
7) echo Saturday ;;
esac
```

תרגיל 7:

תכנן פונקציה או סקריפט בשם lines המקבל שלושה ארגומנטים: שני מספרים שלמים m , n , ושם של קובץ טקסט. התוכנית תפלוט את כל השורות בקובץ שמספרן בין m ל- n (כולל).
הדרכה: תוכל להתבסס על הפקודות head, tail.

פיתרון:

```
#!/bin/bash
if
  test $# != 3
then
  echo "$0: wrong number of arguments!"
fi
file=$1 ; first=$2 ; last=$3
if
  ! test -f $file
then
  echo "$file: no such file!"
fi
nlines=$(cat $file | wc -l)
ntail=$((nlines-first+1))
nhead=$((last-first+1))
tail -$ntail $file | head -$nhead
```

פתרון שני, יותר קצר, המתבסס על awk:

```
#!/bin/bash
if
```



```
    test $# != 3
then
    echo "$0: wrong number of arguments!"
fi
file=$1 ; first=$2 ; last=$3
## we should check that $2 and $3 are integers!
## is ${0<=$2}==1 is good for that
if
    ! test -f $file
then
    echo "$file: no such file!"
fi
awk "$first<=NR && NR<=$last {print \$0}" $file
```

תרגיל 8:

כתוב פקודה פשוטה אשר הפלט שלה יהיה מספר המשתמשים שהם תלמידי שנה א' בעלי חשבונות במחשב moon.netanya.ac.il.

רמז: wc.

פיתרון:

```
ls /home/cs/stud97 | wc -l
```

תרגיל 9:

כתוב פונקציה או סקריפט של bash בשם count אשר הפעלתה תתן את המספר הכולל של קובצי ההפעלה (פקודות הקיימות במסלול PATH).

פיתרון:

```
#!/bin/bash
total=0
for dir in $(echo $PATH | tr : ' ')
do
    for f in $dir/*
    do
        if test -x $f
        then
            total=$((total+1))
        fi
    done
done
echo "You have $total executable files in your PATH"
```

תרגיל 10:

תכנן פונקציה או סקריפט בשם `addalias` המקבלת שני ארגומנטים: הארגומנט הראשון יהיה שם של קיצור, והארגומנט השני יהיה הפקודה שהקיצור מייצג. מטרת הסקריפט תהיה ליצור את הקיצור ובנוסף לכך להוסיף את פקודת `alias` המתאימה לקובץ `.bash_aliases`. בכדי שיהיה קבוע.

פיתרון:

```
addalias ()
{
    local name=$1 value="$2"
    echo alias $name=\'$value\' >> ~/.bash_aliases
    eval alias $name=\'$value\'
    alias $name
}
```

תרגיל 11:

הסבר על ידי משפט אחד מה בדיוק מתבצע על ידי:

```
du -S /home/cs/stud97 | sort -rn | less
```

תרגיל 12:

הסבר על ידי משפט אחד מה בדיוק מתבצע על ידי:

```
w -h | awk '{print $1}' | sort | uniq | wc -l
```

פיתרון:

פלט: מספר המשתמשים המחוברים כרגע למערכת.

תרגיל 13:

הסבר על ידי משפט אחד מה בדיוק מתבצע על ידי:

```
set $(grep user53 /etc/passwd | cut -d: -f 5) ; echo $1
```

פיתרון:

פלט: השם המלא של user53.

תרגיל 15:

כתוב סקריפט בשם hello20 המציג את המשפט "Hello, world" 20 פעם על המסך.

פיתרון:

```
#!/bin/bash
i=1
while
  test $i -le 10
do
  echo "Hello, world"
  i=$((i+1))
done
```

תרגיל 16:

כתוב סקריפט בשם mail המקבלת כפרמטר שם קובץ ושולח אותו בדואר (mail) לכל המשתמשים העובדים באותו רגע על המחשב.
פיתרון:

```
#!/bin/bash
if
  test $# -ne 1
then
  echo "One file name please!"
  exit 1
fi
file=$1
if
  ! test -f $file
then
  echo "$file: no such file"
  exit 2
fi
for user in $(w -h | awk '{print $1}' | sort | uniq)
do
  mail $user < $file
done
```

תרגיל 19:לפניך סקריפט שנקרא `lo.getname`:

```
#!/bin/bash
user=$1
grep $user /etc/passwd | {
  IFS=: read f1 f2 f3 f4 f5 f6 f7
  echo $f5 | cut -d, -f 1
}
```

נסה להסביר בקצרה מה תבצע הפקודה: `lo.getname u3204567`.
למטרה זו קרא את דפי העזרה של הפקודות `read`, `cut`.

פיתרון:

שלב ראשון: נקבל את השדה החמישי של שורת המשתמש. השדה החמישי של שורת המשתמש בקובץ `/etc/passwd` מכיל פרטים שונים אודות המשתמש. פרטים אלה מופרדים על ידי פסיקים. הפרט הראשון הוא השם המלא של המשתמש. פרט זה מתקבל על ידי שימוש בפקודה `cut` (כאשר תו ההפרדה הוא פסיק).

תרגיל 20:

על סמך הפונקציה הקודמת והפקודה last, כתוב סקריפט בשם mylast המקבל כארגומנט יחיד מספר שלם n, ומציג את רשימת n המשתמשים האחרונים שהתחברו למחשב בפורמט הבא:

```
host> mylast 10
u3201234   Avi Rozen           Fri May 1 13:25 - 17:03
u2804216   Ronit Cohen          Fri May 1 13:25 - 17:03
u5206021   Eran Levy            Fri May 1 13:25 - 17:03
. . . . .
user number 10 . . . . .
```

פיתרון:

פלט אופייני של הפקודה "last -R -n" (n הוא מספר המשתמשים האחרונים), נראה כך:

```
root      tty5           Sun Jun 7 17:36   still logged in
galit     tty2           Sun Jun 7 17:12   still logged in
root      tty1           Sun Jun 7 17:10   still logged in
reboot    system boot    Sun Jun 6 12:43
u3401234  tty9           Sat Jun 5 23:47 - 01:04 (01:16)
dany      tty7           Sat Jun 5 23:46 - 01:04 (01:17)
u6712345  tty7           Sat Jun 5 23:40 - 23:52 (12:00)
wtmp begins Wed Jun 3 02:30:07 1998
```

המשתמשים מופיעים על פי סדר הכניסה למחשב. המשתמש האחרון מופיע ראשון. זה בדיוק מה שרצוי לנו. השימוש בשדות של awk אינו אפשרי משום שלמשל בשורה 4 מופיעות שתי מילים בשדה של המסוף לעומת מילה אחת בשורות אחרות. כאן צריך להתבסס על פורמט של תווים.

```
#!/bin/bash
# The output structure of the "last -n -R" command is:
# 1-8:      login name
```



```
# 10-21:  tty description
# 23-end: login time
if
    test $# != 1  -o  ${$1>0} != 1
then
    echo "Usage: $0 num ..."
    exit 1
fi
last -$1 -R | {
    while read line
    do
        user=$(echo "$line" | cut -c1-8)
        name=$(getname $user)
        time=$(echo "$line" | cut -c23-)
        printf "%-12s %-20s %s\n" $user "$name" "$time"
    done
}
```

תרגיל 21:

מטרת התרגיל הבא היא לתכנן מחדש את הפקודה `rm`:
עליך לכתוב סקריפט בשם `rm` אשר

א. יקבל מספר כלשהו של ארגומנטים שהם שמות של קבצים או ספריות.

ב. הסקריפט יעביר את כולם לספריה נסתרת בשם `.recycled` ("פח אשפה") שתמצא בספריית הבית של המשתמש (יש לייצור אותה קודם).

ג. -l יציג את תוכן "פח האשפה".

ד. -e ירוקן את "פח האשפה".

פיתרון:

```
#!/bin/bash
if
    test $# -eq 0
then
    echo "Usage: must supply at least one arg!"
    exit 1
fi
RECYC=$HOME/.recycled
case $1 in
-1) ls -l $RECYC ;;
-e) /bin/rm -f -R $RECYC/* ;;
*) for x in "$@"
    do
        if
            ! test -e $x
        then
            echo "$x: No such file or directory!"
            echo "Nothing removed"
            exit 2
        fi
    done
    mv $* $RECYC ;;
esac
```

יש לשים לב לדברים הבאים:

א. בכדי שהפקודה החדשה `rm` תעבוד יש לשים את הסקריפט בספרייה `$HOME/bin` ולדאוג לכך שספרייה זו תופיע

במשתנה PATH לפני הספרייה /bin) (אשר בא נמצאת הפקודה rm הישנה). מומלץ להתקין סקריפט זה בחשבון האישי של כל אחד, בכדי לא לאבד קבצים חשובים על ידי שימוש שגוי בפקודה rm הישנה (לא ניתן לשחזר קבצים כאלה!).

ב. הפקודה פועלת גם על ספריות שלמות (על כל תתי-ספריותיהן). מאפיין זה אינו קיים בפקודה rm הישנה משום הסכנה שבדבר. אך במקרה שלנו לא קיימת שום סכנה! הספרייה פשוט עוברת לתוך הספרייה ..recycled.

ג. אפשר להוסיף שיפורים נוספים על ידי הוספת פקודות בקובץ .bash_logout לשם ריקון תקופתי של סל המיחזור.

תרגיל 23:

כתוב סקריפט המקבל כארגומנטים שני שמות של קובצי טקסט ומציג את ההפרש בין מספר המילים בשני הקבצים. העזר בפקודה

.wc

פיתרון:

להלן גוף הסקריפט ללא בדיקת ארגומנטים:

```
#!/bin/bash
a=$(cat $1 | wc -w)
b=$(cat $2 | wc -w)
echo "The difference is: ${a-b}"
```

תרגיל 24:

הסבר באופן מלא מה מתבצע על ידי הסקריפט הבא (שנקרא לו concordance):

```
#!/bin/bash
cat $* |
tr "[A-Z]" "[a-z]" |
tr -cs "[a-z']" "\012" |
sort |
uniq -c |
sort -nr +1d |
pr -w80 -4 -h "Concordance for $*"
```

פיתרון:

הארגומנטים לסקריפט זה הם שמות קובצי טקסט. הסקריפט יחזיר את רשימת כל המילים באלפבית האנגלי שמופיעות בקבצים הנ"ל, בסדר מילוני, כשלצד כל מילה יופיע מספר ההופעות שלה בקבצים.

תרגיל 25:

כתוב סקריפט בשם lookfor אשר יקבל שני ארגומנטים: שם של ספרייה dir ומילה word. הסקריפט יחזיר את רשימת כל הקבצים בספרייה dir (כולל תתי-ספריות) אשר בהם מופיעה המילה word. מתחת לשם כל קובץ יופיעו כל השורות בקובץ בהן המילה word מופיעה.

הדרכה: למד להשתמש בפקודה !find
פיתרון:

```
#!/bin/bash
#
# Usage: lookfor <directory> <-/+days> <word> <word> ...
# lookfor - look for all files in the
#           specified directory hierarchy
#           modified within a certain time,
#           and pass the resulting names to
#           grep to scan for a particular pattern.
# Temporary file to store find results
TEMP=/tmp/lookfor$$
# Clean up if we're interrupted
trap 'rm -f $TEMP; exit' 0 1 2 15
# Find all files in the specified directory
# that were modified within the specified time
find $1 -mtime $2 -print > $TEMP
# shift the directory name and modification time off the command line
shift; shift
# Look for each word (ignoring case) in the result of the find
for word
do grep -i "$word" 'cat $TEMP' /dev/null
done
```

תרגיל 26:

כידוע, חלק גדול מהפקודות של Unix הם למעשה סקריפטים שנמצאים ברובם הגדול בספריה `/usr/bin`, ובספריות נוספות אחרות. כתוב סקריפט בשם `scripts` שמציג את רשימת כל הסקריפטים במערכת הכלולים במשתנה הסביבתי `PATH`.

הצעה: התבסס על הפקודה `file`.

פיתרון:

```
#!/bin/bash
total=0
for dir in $(echo $PATH | tr : ' ')
do
  for f in $dir/*
  do
    if
      test -x $f & file $f | grep "shell script"
    then
      total=$((total+1))
    fi
  done
done
echo "-----"
echo "You have $total shell scripts in your PATH"
```


תרגיל 28:

כתוב סקריפט בשם nom המקבל ארגומנט יחיד שהוא ביטוי רגולרי (לתאור רשימת קבצים) והמציג את רשימת כל הקבצים בספריה הנוכחית אשר אינם תואמים את הביטוי הרגולרי.

```
host> ls
prog1.exe  prog1.o  readme.txt  prog2.exe
stam.txt   foo.doc  bar.exe     prog2.o
host> nom *.exe
prog1.o    readme.txt  stom.txt
foo.doc    prog2.o
host> mv $(nom *.exe) /tmp/my-garbage
```

פיתרון:

```
#!/bin/bash
# must have at least one argument,
# and all have to be in current directory:
case "$*" in
  "")    echo Usage: $0 pattern 1>&2
        exit ;;
  */*)  echo "$0 quitting: I can't handle '/'s." 1>&2
        exit ;;
esac
for x in *
do
  match=no
  for y in "$@"
  do
    if
      test $x = $y
    then
      match=yes
      break
    fi
  done
  if
    test $match = no
  then
    echo $x
  fi
done
```