פרק #1 - הקדמה

מערכת הפעלה (מ.ה., Operating System)
היא תכנית
המשמשת כמתווכת בין המשתמש לחומרה.

מטרתה: לספק סביבה בה המשתמש יוכל להריץ תכניות *בנוחות*, *וביעילות*.

עליה להבטיח את הפעולה התקינה של המחשב; למנוע מתכניות המורצות לפגוע בתקינות המערכת או זו בזו, בעזרת מנגנונים המסופקים ע"י החומרה\ארכיטקטורה.

ראוי שהיא תסייע לנצל את משאבי החומרה במידה המיטבית.

1.1 מהי מ.ה.?

ניתן לחלק את 'המחשב' בגסות לארבעה מרכיבים: חומרה, מ.ה., אפליקציות, משתמשים.

החומרה = מעבד, זיכרון, ציוד קלט\פלט (ק\פ) מספקת את משאבי העיבוד הבסיסיים.

האפליקציות = (מעבד תמלילים, דפדפן, מהדר) קובעות את השימוש שנעשה בחומרה כדי לספק את צרכי המשתמש.

מ.ה. מתווכת ומבקרת את השימוש בחומרה ע"י האפליקציות השונות, עבור ציבור המשתמשים.

מ.ה. מהווה מקצֶה משאבים (resource allocator): במערכת מחשב ישנם משאבי חומרה ותכנה רבים. מ.ה. מנהלת משאבים אלה. היא מקבלת בקשות לשימוש במשאבים, ועליה להקצותם לתכניות ולמשתמשים ביעילות ובהוגנות.

פרספקטיבה מעט שונה: מ.ה. היא תכנית בקרה\שליטה אשר מנהלת\שולטת את\על תכניות המשתמשים, כדי למנוע שימוש שגוי במערכת המחשב.

מ.ה. קיימות שכן הן מהוות את הפתרון הטוב ביותר הקיים לסוגיית תפעול המחשב, כך שצורכי *המשתמשים* יסופקו באופן המיטבי ע"י *חומרת המחשב*, אשר תריץ *תכניות* שחולקות משאבים אותם יש לנהל.

מ.ה. דומה לְממשלה: אין לה תפקיד בפני עצמה, כל תפקידה הוא לספק סביבה בה אחרים (האפליקציות) יוכלו לפעול כהלכה.

בעולם אידיאלי, בו תכניות היו תקינות, ומסוגלות לנהל בעצמן את משאבי המערכת, לא היה צורך במ.ה.

אין הגדרה חד משמעית מהי מ.ה..
שתי הגדרות קיצוניות בתפיסתן:

א. מ.ה. היא כל מה שהמשווק מספק לך עת אתה רוכש מ.ה.
ב. מ.ה. היא התכנית שרצה במחשב *כל העת* (אשר לעתים נקראת *ליבת מ.ה.* o.s. kernel).

שתיים ממטרותיה העקרוניות (אך חלקית מנוגדות) של מ.ה. הן:

א. להקל על המשתמשים (בפרט, ממשקים גרפיים מממשים מטרה זאת).
ב. למקסם את יעילות המערכת.

נציג תסריט המתאר את פעולת המחשב בקטע זמן כלשהו:

א. מ.ה. מתזמנת (schedules) תכנית א', כלומר מאפשרת לה לרוץ.
ב. תכנית א' מורצת, כלומר המעבד מבצע את פקודותיה. מ.ה. אינה מעורבת.
ג. שעון המערכת שולח פסיקה למעבד. המעבד פונה לבצע את שגרת הטיפול בפסיקה (interrupt handler), שהינה פונ' של מ.ה.. ביצוע תכנית א' מושהה עד סיום ביצוע הטפלית.
ד. שגרת הטיפול בפסיקה מורה כי יש להחליף את התכנית המורצת ע"י המעבד. מ.ה. בוחרת אפליקציה אחרת להריץ, ב', ולשם כך מבצעת החלפת הקשר (context switch).
ה. אפליקציה ב' מורצת. אחרי זמן מה היא מבצעת קריאת מערכת (ק.מ. system call), כדי לקרוא נתון מהמקלדת (או מקובץ).
ו. ק.מ. גורמת למלכודת (trap) למ.ה. המתעוררת שוב לפעולה כדי לבצע את הקלט. אפליקציה ב' נשלחת בינתיים להמתין.
ז. מ.ה. מתזמנת אפליקציה חדשה להרצה (ע"מ שהמעבד לא יושבת) ...
ח. הקלט של אפליקציה ב' הסתיים. ציוד הקלט שולח פסיקה המורַה על כך. לכשיגיע תורה, אפליקציה ב' תוכל לרוץ שוב. מה קורה בינתיים עם ציוד הק\פ?

1.2 מערכות אצווה, ריבוי תכניות ושיתוף זמן

בסעיף זה נסקור את האבולוציה שעברו מערכות המחשב מימי קדם ועד ימינו.

א. הרצת תכנית\ג'וב בודד\ת

במחשבים בשנות החמישים המשתמש (או המתכנת) לא בא במגע עם המחשב. המשתמש הכין על-גבי כרטיסים ג'וב = תכנית שיש להריץ + נתונים. הג'וב נמסר למפעיל שהריצו (עת הגיע תורו של הג'וב) ומסר למשתמש את הפלט שכלל את תוצאת ההרצה + הצגת מצב הזיכרון והאוגרים בסיום הריצה (לטובת דבגינג).

מ.ה. באותם ימים הייתה, על-כן בסיסית ביותר: רק תכנית אחת הורצה במחשב בכל נקודת זמן, וכל מה שצריך היה הוא לאפשר לה לרוץ.

ב. הרצת קבוצת ג'ובים

כדי לייעל את השימוש במחשב, במקום שכל ג'וב ייטען בנפרד (דבר המצריך זמן), קובצו מספר ג'ובים יחד והורצו כקבוצה\אצווה (batch). כאן כבר צריך היה לדאוג גם למעבר תקין מג'וב לג'וב, ולכן מ.ה. צעדה צעד קדימה.

עדיין בזיכרון שכנה, והורצה רק תכנית בודדת.

המשמעות: מכיוון שבהרבה מקרים משך ביצוע ק\פ ארוך במידה ניכרת ממשך החישוב (במחשב בו המעבד מבצע כמה אלפי פעולות בשנייה, אך קורא הכרטיסים קורא עשרים כרטיסים בשנייה) אזי המעבד מושבת אחוז ניכר מהזמן.

עם המצאת הדיסק המגנטי (אמצע שנות החמישים) קריאת כל ג'וב נעשתה מהירה יותר, ומה שחשוב לא פחות: ניתן היה לגשת ישירות לג'וב רצוי (ולא סדרתית ג'וב אחר ג'וב, כמו בקורא כרטיסים). מייד נראה מדוע יש לכך חשיבות.

ג. ריבוי תכניות (Multiprogramming)

עד כה רק תכנית בודדת הוחזקה בזיכרון בכל נקודת זמן, ועל כן ניצולת המעבד (וציוד ק\פ הייתה נמוכה).

הרעיון בריבוי תכניות: מ.ה. שומרת בזיכרון כמה ג'ובים בו זמנית (ומעבר להם ג'ובים נוספים מוחזקים בדיסק). המערכת מתזמנת ג'וב כלשהו להרצה; עת הג'וב מבצע ק\פ (ועל-כן המעבד מושבת) מ.ה. מתזמנת ג'וב אחר לרוץ.

על דרך השלילה: מ.ה. אינה גוזלת את המעבד מג'וב שמורץ. ג'וב אחר יורץ רק עת הג'וב שהורץ עד כה ויתר על המעבד מרצונו.

במערכת כזאת מ.ה. מבצעת שתי החלטות משמעותיות:

- תזמון המעבד (תזמון קצר מועד): עת ג'וב פינה את המעבד, מי יהיה הג'וב שיזכה בו (נחזור לסוגיה זאת בפרק #6).
- תזמון ג'ובים (תזמון ארוך טווח): עת ג'וב סיים לרוץ, ויצא מהזיכרון, מי יהיה הג'וב שיטען לזיכרון (וכך יזכה באפשרות להיות מתוזמן לרוץ).

ד. מערכות שיתוף זמן (Time-Sharing Systems)

כפי שאמרנו, מערכות ריבוי תכניות מאופיינות בכך שג'וב אוחז במעבד עד אשר הוא מוותר עליו מרצונו. בניגוד לכך במערכות שיתוף זמן (נקראות גם ריבוי משימות multitasking) המעבד מבצע 'במקביל' מספר ג'ובים\משימות תוך שהוא מחליף ביניהם *בתדירות גבוהה;* במידת הצורך, ע"י גזילת המעבד מתהליך שאחז בו פרק זמן כלשהו (ביוניקס 01. השנייה).

התוצאה: כל משתמש זוכה לאילוזיה כאילו הוא עובד לבד על מחשב (איטי יותר).

היבט נוסף של שיתוף זמן: hands-on= המשתמשים עובדים מול התכניות המורצות על-ידם בצורה אינטראקטיבית: מזינים קלטים ומקבלים פלטים.

אם עד כה דנו בעיקר בניצולת המערכת, אזי במערכות שיתוף זמן מדד נוסף להערכת ביצועי המערכת הופך משמעותי: זמן התגובה (response time) = משך הזמן שעובר עד שהתכנית שולחת פלט ראשון למשתמש. (נחזור לנושא בפרק #6).

במערכות שיתוף זמן המושג המקובל אינו הג'וב כי אם התהליך (process) = תכנית שנטענה לזיכרון ומבוצעת (מעת לעת) ע"י המעבד (לצד תכניות אחרות).

לכל משתמש במערכת, בכל נקודת זמן, יש תהליך אחד לפחות.

ביוניקס, עת מערכת ההפעלה עולה היא מייצרת תהליך בשם init (התהליך #1 במערכת), אשר רץ כל הזמן. עת אתם מבצעים login יוצר עבורכם init תהליך (ילד) שהינו ה- shell שלכם. בכל פעם שאתם מריצים תכנית, ה- shell מייצר (מוליד) תהליך חדש שיריץ את התכנית המבוקשת (emacs, ls, Firefox).

ה. נושאים שנפגוש לאורך הדרך

ראינו שבמערכות ריבוי-תכניות ובמערכות שיתוף-זמן כמה תכניות\תהליכים מוחזקות\ים בזיכרון בו-זמנית. על כן יש להגן על כל תהליך מפני עמיתיו. פרק #9 דן בניהול הזיכרון באופן כללי, ובנושא זה בפרט.

בשתי המערכות מספר תהליכים מוחזקים בזיכרון בו זמנית. מכיוון שגודלו של הזיכרון מוגבל, יש מגבלה על מספר התהליכים שהזיכרון יכול להכיל במקביל בכל נקודת זמן. בעיקר במערכות שיתוף זמן ייתכן מצב בו המשתמשים יוצרים מספר רב של תהליכים (בלי שלמערכת יש על כך שליטה). הפתרון: רק חלק מכל תהליך יוחזק בזיכרון בכל נקודת זמן, והיתר יישמר\יוותר בדיסק אליו נתייחס כאל הרחבה של זיכרון. השיטה נקראת זיכרון מדומה (virtual memory) ונדון בה בפרק #10.

מערכת המחשב, בפרט כזו של שיתוף זמן עושה שימוש אינטנסיבי בדיסק. פרקים #11 עד #14 ידונו בניהול הדיסק, ובמערכת הקבצים השוכנת עליו.

עת מספר תהליכים מורצים במקביל הם עשויים לשתף פעולה אלה עם אלה לשם השלמת משימה כלשהי. פרק #5 מציג את מושג הפתיל\תהליכון (thread) שהינו 'אח רזה' של התהליך, המיועד בעיקר למצבים בהן נרצה להשלים משימה בעזרת כמה 'תהליכים' שירוצו במקביל (וישתפו פעולה).

תהליכים כנ"ל, המשתפים פעולה זה עם זה, צריכים, במקרים רבים, להחליף נתונים זה עם זה. פרק #4 יציג בהרחבה את נושא התהליך, ואת האופן בו תהליכים עשויים לתקשר זה עם זה ביוניקס.

במערכת בה מספר תהליכים חולקים ביניהם מספר משאבים יש חשש מחסימות הדדיות, ומתקלות אחרות שמקורן בשימוש המקבילי במשאבים (המדפסת תדפיס במעורבב שתי הדפסות שונות). פרקים #7, #8 מציגים נושאים אלה, וכיצד נשמר מתקלות.

1.3 שירותים אותם מספקת מ.ה.

בסעיף זה אמנה שירותים , חלקם הוזכרו כבר קודם לכן, המסופקים ע"י מערכת ההפעלה וע"י תכניות השרות הנלוות:

א. ממשק משתמש (user interface, UI)
מאפשר הזנת פקודות למ.ה. בצורה טקסטואלית = באמצעות הקלדת הפקודה כשורת פקודה (command-line), או באופן גראפי (graphical user interface, GUI) באמצעות חלונות, תפריטים, הצבעה ע"י סמן, והקלדת פרמטרים.

ממשק טקסטואלי עשוי להיות אינטראקטיבי, או כקובץ אצווה (batch) שפקודותיו מבוצעות בזו אחר זו. (ביוניקס: tcsh commands.txt עבור commands.txt שהינו קובץ טקסט רגיל הכולל סדרת פקודות [אין צורך בטיפול בהרשאות]).

ביוניקס, ממשק המשתמש אינו חלק מגרעין מערכת ההפעלה, אלא הינו תכנית נלווית.

ב. הרצת תכניות (program execution): טעינת תכנית (המצויה בדיסק) לזיכרון, ביצוע כל ההכנות שיאפשרו את הרצתה, ביצועה, עם סיומה: קבלת ערך ההחזרה שלה, [לדוגמה, ב- tcsh: $? echo), ומִחזור כל המשאבים שהוקצו לה.

זוהי משימה מובהקת של גרעין מ.ה.

ג. ביצוע ק\פ: תכנית רצה מבצעת ק\פ. האופן המדויק בו יש לבצע את הק\פ הוא תלוי ציוד (מסך, דיסק, CD). מטעמי פשטות, יעילות ובטיחות תכנית המשתמש לא רשאית לבצע ק\פ בעצמה. היא מבקשת שירות זה ממ.ה. (באמצעות קריאת מערכת, system call).

ביצוע ק\פ כרוך בהפעלת גרעין מ.ה., כמו גם מנהלי התקנים שהינם חלק ממ.ה. אך לא מהגרעין.

ד. טיפול במערכת הקבצים (file-system manipulation): כל הפעולות על מערכת הקבצים (יצירת קובץ, מחיקתו, חיפושו, יצירת מדריך, שליפת ועדכון מידע אודות קובץ או מדריך) מבוצעות רק ע"י מ.ה.. המשתמש מבקש שירות זה מהמערכת (באמצעות ק.מ.); אופן מימוש השרות שקוף לו.

ה. תקשורת בין תהליכים (Inter-process communication IPC): הזכרנו שתהליכים עשויים לרצות להעביר ביניהם מידע (הם מבצעים משימה משותפת, או: תהליך א' [דפדפן] מבקש מידע השמור בידי תהליך ב' [שרת]), בין אם הם רצים באותה מערכת ובין אם לאו.

פרק #4 מציג כלים רבים ומגוונים לתקשורת בין תהליכים כפי שקיימים ביוניקס.

ו. מעקב אחר שגיאות (error detection): תכנית, כפי שאולי שָׁמעתם, עלולה לשגות במגוון רחב של אופנים; גם החומרה עלולה לשגות (החל בנפילת מתח, וכלה בכישלון בכתיבה לדיסק, למדפסת, לרשת). מ.ה. צריכה לנטר שגיאות אלה, ולהגיב להן.

פרק #2, ויותר מכך פרק #4, מציג את נושא הסיגנל (signal) שהינו אחת הדרכים באמצעותה מ.ה. יוניקס מאותתת לתכנית על שגיאה שהתכנית ביצעה.

מ.ה. מספקת כלים לדיבגינג שיאפשרו לצמצם את כמות השגיאות.

ז. הקצאת משאבים (resource allocation): הזכרנו משאבים רבים שבאחריות מ.ה. להקצות: זיכרון, זמן מעבד, שטח בדיסק, מדפסת, אפשרות גישה לערוצי תקשורת, ועוד רבים.

באחריות מ.ה. להקצות את המשאבים על-פי קריטריונים שונים (ניצולת ציוד, חשיבות המבקש, הוותק שלו) תוך הקפדה שלא 'לקפח' במלים אחרות להרעיב אף תהליך.

ח. חיוב (accounting): רישום מי צרך מה, לשם חיוב ולשם שיפור הביצועים.

ט. הגנות ובטיחות (protection and security): הגנה על הזיכרון, על הדיסק, הן מפני תכניות המורצות במערכת, והן מפני פריצה מבחוץ.

1.4 קצת על יוניקס (UNIX) ולינוקס (Linux)

תחילת דרכה של יוניקס במעבדות Bell (חברת בת של AT&T), עת בשנת 1969 קן תומפסון שיכתב ושיפר מ.ה. קודמת.

בהמשך, חבַר לתומפסון גם דניס ריצ'י ויחד, בשנות ה- 70' השניים קידמו מאוד את מ.ה..

בפְרט, ריצ'י הגדיר\המציא את שפת C כשפה שתתאים לכתיבת מ.ה.. ריצ'י ותומפסון כתבו לשפה מהדר, ואחר שיכתבו את מרבית מ.ה. בשפת C.

כתיבת רוב מ.ה. יוניקס בשפת עילית הייתה מעשה חדשני, שכמובן תרם לפורטאביליות של המערכת.

בשנת 74' זכו ריצ'י ותומפסון בפרס טיורינג על עבודתם.

בשנות ה- 70' הופצה מ.ה., כולל הקוד שלה, לאוניברסיטאות שונות; שם היא שוכתבה, שופרה הורחבה, וזכתה לפופולאריות.

UNIX Version 7 הייתה הגרסה המתקדמת ביותר שהופצה באקדמיה ובתעשייה בשנות ה- 80' כגרסה אחת ויחידה של מ.ה. יוניקס.

בשלב זה חל פיצול בעולם היוניקסי:

- חברת AT&T פיתחה את גרסה 7 לכדי סדרה של מ.ה. ובראשן System V שכללה מספר הפצות (זיכרון משותף, סמפורים, תורי הודעות מקורם בדיאלקט זה).
- במקביל, פותח דיאלקט מתחרה ע"י מתכנתים מאוניברסיטת ברקלי. הדיאלקט נקרא: BSD = Berkley Software Distribution, וגרסה 4.x בו היא המוכרת ביותר. (הציגה: זיכרון מדומה, paging, תקשורת באמצעות TCP/IP, תושבת [socket], ותמיכה משופרת בשפות תכנות). מערכת ההפעלה SunOS התבססה על גרסה זאת.

התוצאה: בשוק קיימים שני דיאלקטים לא תואמים ונפוצים של יוניקס, ומתנהלות UNIX wars בין המתחרים.

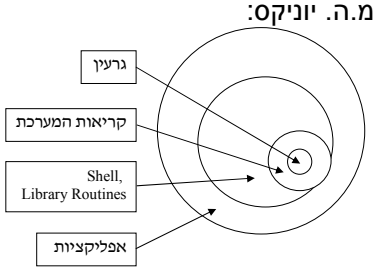
באמצע שנות ה- 80' החליט ה- IEEE (המכון למהנדסי חשמל ואלקטרוניקה) להגדיר תקן אחיד ליוניקס. התקן נקרא POSIX = Portable OS Interface (הסיפא: ix מציינת שהוא ממורשת Unix). התקן הגדיר ממשק שמ.ה. אמורה לספק למתכנת:
API = Application Program Interface.
העיקרון בהגדרת התקן היה חיתוך של שני הדיאלקטים.

בשנת 1989 הוצאה System V Release 4 (SVR4) שהייתה תוצאה של שיתוף פעולה בין שני הגורמים הדומיננטיים בעולם היוניקס: AT&T ו- Sun (שצמחה בברקלי). מ.ה. Solaris של Sun מבוססת על SVR4.

בראשית שנות ה-90' שוב חל פיצול בעולם היוניקס: HP, DEC, IBM הקימו את OSF = Open Software Foundation, בתגובה AT&T הקימה את UI = UNIX International, ושוב נוצרו שני דיאלקטים מתחרים.

כיום שולטים בעולם היוניקס בעיקר Solaris (של Sun, צאצא של SVR4), ולצדו: HP-UX ו- AIX של IBM. גם Mac OS X הוא מערכת יוניקסית (מבוססת על גרסה אקדמית בשם Mach של יוניקס ועל FreeBSD אשר כשמה כן היא).

שרטוט סכמתי של מ.ה. יוניקס:



חידושים שהופצו בעזרת יוניקס:

א. מ.ה. הכתובה ברובה בשפה גבוהה.
ב. התייחסות אחידה ומינימליסטית לכל הקבצים כאל רצף של בתים.
ג. היררכיה בלתי מוגבלת של מדריכים.
ד. הפיכת ה- shell לתכנית שמורצת ברמת המשתמש, ומריצה פקודות: כ"א הינה תכנית נפרדת.
ה. אוסף אחד של פקודות משמש הן לעבודה אינטראקטיבית, והן ל- shell scripts.
ו. שימוש ב- pipe (צינור) לתקשורת בין תהליכים.
ז. מגוון כלי IPC (מעבר ל- pipe).
ח. שימוש בביטויים רגולריים.
ט. שימוש בשפת C.
י. הכנסת TCP/IP למחשבים קטנים.
יא. תיעוד on-line רב, וקוד פתוח.

לינוקס (Linux)

לינוקס אינה מערכת יוניקס 'רשמית', אלא מערכת UNIX-like.

לינוקס נולדה בשנת 1991, עת לינוס טורוַאלְדס (סטודנט פיני בן 21) כתב (ליתר דיוק שיכתב) גרעין מ.ה. למעבד 30386, מעבד 32 הסיביות הראשון של אינטל למחשבי PC.

גרסה 0.01 כללה 10^4 שורות קוד, הופצה באופן חופשי ברשת, וכך שופרה והורחבה ע"י מתכנתים רבים.

המערכת, שהחלה כתת-מערכת יחסית מצומצמת של יוניקס, הלכה וכיסתה יוצר ויותר תכוניות.

ב-1994 יצאה גרסה 1.0.0 שכללה 176K שורות קוד, והתאימה למעבדי אינטל 386 בלבד.

ב- 1996 הופצה גרסה 2.0.0 שכללה 1.8M שורות, והתאימה למגוון רחב של מעבדים.

ב- 2003 הופצה גרסה 2.6.0 שכללה 6M שורות.

לגרעין מ.ה. צורפו מרכיבים שנלקחו ממערכות יוניקסיות אחרות על-מנת לקבל מ.ה. שלמה: בפרט מרכיבים מפרויקט ה- GNU (של ריצ'ארד סטולמאן) = פיתוח פַתוח של כלי יוניקס (לדוגמה: gcc = GNU C Compiler). (גם BSD תורמת מרכיבים למ.ה., ויש מרכיבים שהינם ספציפיים להפצות שונות). על-כן המדייקים מכנים את מ.ה. השלמה (בניגוד לגרעין מ.ה.) GNU/Linux.

הפצה של לינוקס (Linux Distribution)

בראשית דרה של לינוקס, כדי לקבל מ.ה. שלמה היה על הלקוח לקבץ ולהתקין מרכיבים שונים מהגורן ומהיקב, ולשלבם יחד בכוחות עצמו (משימה מסובכת למדי).

רעיון *ההפצה* הוא למסור לידי הלקוח 'חבילה' שלמה הכוללת:

א. גרעין לינוקס שהינו (כמעט) אחיד בכל ההפצות.
ב. כלים וספריות GNU – די אחידות.
ג. תוכנות נוספות.
ד. תיעוד + קוד המערכת.
ה. סביבת שולחן עבודה המתבססת על תכנת ניהול חלונות (במקרים רבים תכנת X של BSD).
ו. כלי התקנה ואדמיניסטרציה שיקלו על בניית מערכת אינטגרטיבית, ובהמשך עדכונה.
ז. מרכיבי בטיחות שונים.

כל הפצה מתחלקת לחבילות (packages). כל חבילה כוללת ישום או שרות (דפדפן, פונטים). חבילה מגיעה מקומפלת ועם תכנת התקנה\הסרה\שדרוג.

אחד ההבדלים בין ההפצות הוא באופן בו הן מקונפגות, ובמאמץ הנדרש לשם קינפוגן.

במקומותינו, כיום, מותקנת ההפצה CentOS: גרסה נטולת תמיכה, מאוד יציבה, חינמית, ולא המתקדמת ביותר (יחסית ל- Fedora) של Red Hat.

רק כדי לסבר את העין: Red Hat 7.1 משנת 2001 כללה $30*10^6$ שורות קוד מקור, שחייבו $8*10^3$ שנות אדם לפיתוחן, והיו צריכות לעלות 10^9 דולרים.

כ: 70% מהקוד נכתב בשפת C, והיתר במגוון של שפות (החל באסמבלר וכלה בליספ). נפח הגרעין היה $2.4*10^6$ שורות, או 8% מכלל מ.ה..

1.5 מניין באנו

במחשבים הראשונים לא הייתה מ.ה.. המשתמש\מתכנת (שתכנת בשפת מכונה) עבד לבד על המחשב, ותכניתו קיבלה את כל השליטה על המכונה.

התכנית הייתה צריכה לטפל בכל רכיבי החומרה, וכללה את כל מנהלי ההתקנים (לקורא\מנקב כרטיסים, מדפסת, סרט נייר\מגנטי).

התכנית רצה עד סיום או תעופה.

בשלב הבא נוספו ספריות של קוד תומך ששולבו בתכנית כדי לבצע משימות סטנדרטיות כגון ק\פ. זהו האב הקדמון של מ.ה.

עם הזמן מהירות הריצה (שבתחילה נמדדה על שעון הקיר באולם המחשב) עלתה (ונמדדה כבר ע"י המחשב), והתקורה בהחלפת ג'ובים נעשתה גבוהה (מדי).

מאנשים שעמדו בתור (כמו לרופא) לזכות בשימוש במחשב, עברו לקופסות כרטיסים שחיכו בערמה, ותופעלו ע"י מפעילים. למתכנת לא הייתה גישה למחשב.

באמצע שנות החמישים, מחשב עולה כשני מילון דולר, ולכן נרצה לעשות בו שימוש אינטנסיבי ככל האפשר, בפרט לא נרצה להשביתו רק משום שהמפעילים צריכים לפרוק ולהעמיס ארגזי כרטיסים (ותוך כדי כך הם גם מאבדים חלק).

המוטיבציה: נרצה למכן את תור הג'ובים, ואת נושא החיוב (זמן ריצה, כמות כרטיסים שנקראו\נוקבו, כמות הדפסות).
התוצאה: 1955 נולדת מ.ה. הראשונה. וכן: המחשב בוחר איזה ג'וב, מאלה שממתינים בתור, ייטען ויורץ.

מתפתח 'מוניטור' תכנית שרצה במחשב טרם שנטענת תכנית המשתמש, טוענת את תכנית המשתמש, מבקרת את ריצתה, מנקה אחריה, מנהלת את החיוב, ופונה להרצת התכנית הבאה.

בשלב זה (שנות השישים) נכנסים גם קומפיילרים (לפורטרן וקובול) שמתרגמים את התכנית טרם הרצתה, ונכללים במ.ה.. בהמשך נוספים עורכים, מנהלי מערכות קבצים, ומ.ה. הולכות ומתפתחות.

בשנות השישים והשבעים, עת מהירות המעבדים גדלה, חלות כמה מהפכות משמעותיות:

א. מחשבים עוברים לעבוד בשיטת חלוקת הזמן (time sharing): כלומר כמה תכניות מורצות ע"י המחשב במקביל. המעבד עובר מעת לעת מתכנית לתכנית.
בתחילה: (שנות ה-60') התכניות מורצות באצווה (batch): הן מוכנות להרצה, כולל הקלט הדרוש להן.
ב. בהמשך (שנות ה- 70'): מתאפשרת עבודה אינטראקטיבית של (לעתים) אלפי משתמשים במקביל על מחשב יחיד (העובד בשיטת חלוקת הזמן).
ג. מתפתח ה-'זיכרון המדומה': לתכנית נדמה שגודל הזיכרון (המוקצה לה) גדול מהפיזי (האמיתי). כך ניתן להריץ תכנית גדולה תוך שימוש בזיכרון קטן.

עד כאן דנו במחשבי mainframe עימם התפתח עולם המחשוב. בשנות השמונים נולדים מחשבי שמונה סיביות, אישיים ראשונים: קומודור 64, אטארי: מחשבים שסופקו כמעט ללא מ.ה.. עם ROM שכלל מפרש בייסיק בסיסי ומיני מ.ה. (טיפול בסיסי בקבצים, ויכולת לטעון תכנית ולהריצה). התכנית קיבלה שליטה מלאה על המחשב.

עם העלייה במגוון רכיבי הציוד שניתן לחבר למחשב עולה הצורך במ.ה..
<u>הפתרון:</u> תכנית boot ששוכנת ב- ROM וטוענת את מ.ה.. לדוגמה: BIOS.

1.6 <u>ולאן הגענו</u>

מחשבי mainframe כיום מאופיינים בכך שהם מסוגלים *לעבוד ברצף* שנים רבות, כך שגם תחזוקתם מתבצעת במקביל למתן שרות.

הם מצופים לתת שרות לאלפי משתמשים, במהירות גבוהה.

אחד ממאפייניהם של מחשבים גדולים היום הוא שהם מריצים מספר מ.ה. במקביל. התוצאה: על מכונה פיזית אחת (המריצה בפועל מ.ה. יחידה הקרויה hypervisor) רצות מספר <u>מכונות מדומות</u> (virtual machine) או <u>מ.ה. אורחות</u> (guest o.s.). (עקרון זה מתפשט כיום גם למחשבים חלשים יותר.)

מאפיין אחר הוא כמות הנתונים המאוחסנת במחשב (קבצים בסד"ג של 10^{12} רשומות), ומהירות הגישה הגבוהה לנתונים (המתאפשרת בעזרת שימוש במעבדים ייעודיים לק\פ).

mainframe נבדלים ממחשבי-על בכך שהראשונים נותנים דגש על מהימנות, קצב ק\פ, והרצת מגוון יישומים במקביל (חישוב משכורות והזמנת טיסות), בעוד האחרונים על קצב עיבוד (חיזוי מזג אוויר) לעתים למשימה מוגדרת.

1.7 <u>מערכות משוכנות Embedded Systems</u>

בקצה השני של הספקטרום ניצבות <u>המערכות המשוכנות</u>: כאן אנו דנים במערכת מחשב (מעבד, זיכרון, ציוד ק\פ) המשוכנת במכשיר כשלהו (שעון דיגיטאלי, מכשיר MP3, מערכת רמזורים, כלי רכב), ומיועדת לבצע חישוב מאוד ספציפי, פעמים רבות, במגבלות זמן אמת.

בשל היות המערכת תפורה ליישום ספציפי היא מותאמת לו ורק לו, כך שגודלה ועלותה ימוזערו, ומהימנותה וביצועיה ימוקסמו.

בד"כ למערכת יהיו משאבי חישוב מוגבלים, זיכרון מצומצם, ויחובר לה ציוד היקפי ספציפי.

תכנת המערכת (firmware) נצרבת על זיכרון לקריאה בלבד (או זיכרון Flash), כך שהיא יחסית יציבה במכשיר.

לדוגמה: בטלפון נייד יש לבצע את המשימות הבאות:

א. קריאת קלט מהמקלדת.
ב. שליחת פלט למסך.
ג. קליטת קלט מהרשת, ושליחתו לאוזניה או למסך.
ד. קליטת קלט מהמיקרופון ושליחתו לאוזניה ולרשת.
ה. קריאת קלט מחיישן המצלמה, ושליחתו לקובץ ע"ג מצע האחסון.
ו. הוספה, מחיקה, חיפוש ברשימת כתובות.
ז. הפעלת משחקים שונים.
ח. ועוד.

מ.ה. עשויה להתנהל באופנים שונים:

א. יש מערכות המריצות לולאה בה בכל סיבוב נקראות הפונ' הדרושות.

ב. יש מערכות שהינן תגובתיות (reactive). במערכות כאלה, אירועים (events) שונים מדווחים למ.ה. באמצעות פסיקות (interrupts). מ.ה. מטפלת בפסיקה. סכמה כזאת מתאימה במיוחד עת יש לטפל באירוע *בזריזות*, ומנגד הטיפול הינו *קצר* (או לפחות חלקו הדחוף קצר).

ג. ישנן מערכות, הדומות יותר למ.ה. במחשבים 'רגילים' בהם המערכת פועלת בשיתוף זמן: מריצה מספר משימות במקביל, תוך שהיא מריצה לסירוגין כמה פקודות מכל משימה\תכנית. עת המערכות מספיק גדולות מותקנת בהן מ.ה. קנויה 'סטנדרטית', אחרת נכתבת מ.ה. בסיסית.

1.8. מערכות זמן אמת Real-Time Systems

מערכות זמן אמת מאופיינות בכך שקיימת בהן מגבלה על פרק הזמן שלוקח למערכת להגיב לאירוע (נעילת גלגלים ברכב או הזזת העכבר). אם המערכת אינה מגיבה לאירוע תוך פרק זמן x היא נכשלה.

(במערכות 'רגילות' זמן טיפול איטי 'יעצבן' את המשתמש, אולם הדבר אינו קריטי.)

נהוג להבחין בין מערכות זמן אמת קשות (hard) בהן אי עמידה בזמנים משמעה קטסטרופה (קוצב לב שלא שולח פולס בזמן, תכנת שח-מט שלא מנפיקה צעד בזמן, ולכן מפסידה במשחק), למערכות זמן אמת רכות (soft) בהן היא 'רק' בגדר פגיעה באיכות הביצועים (תגובה לא די מהירה לקלט המשתמש במשחק מחשב, או עדכון מעט מושהה של מערך טיסות).

במערכת הפעלה 'סטנדרטית' העובדת בשיתוף זמן עמידה בתנאי זמן אמת קשים בד"כ אינה אפשרית, והשאלה היא בד"כ איזה שירות תיתן המערכת בשעת עומס רב למערכות זמן אמת רכות.

אמרנו, שמערכות זמן אמת (קשות) צריכות להגיב תוך פרק זמן x. מנגד, אם הן עומדות בכך אין עניין בשיפור ביצועיהן, ולכן נהוג לומר שבמערכות אלה לא *הביצועים* (performance) הם העיקר אלא *יכולת הניבוי* (predictability) כמה זמן ייקח להגיב. (כמובן ששיפור בביצועים יאפשר לבצע חישוב מורכב יותר במגבלות הזמן שהוצבו.)

בקורס שלנו (כמעט ו)לא נתייחס למערכות זמן אמת.