מערכות הפעלה - הרצאה 4

שרון מלטר, אתגר 17

7 ביולי 2024

נמשיך את פרק 2.

1 *Chapter 2 Cont.*

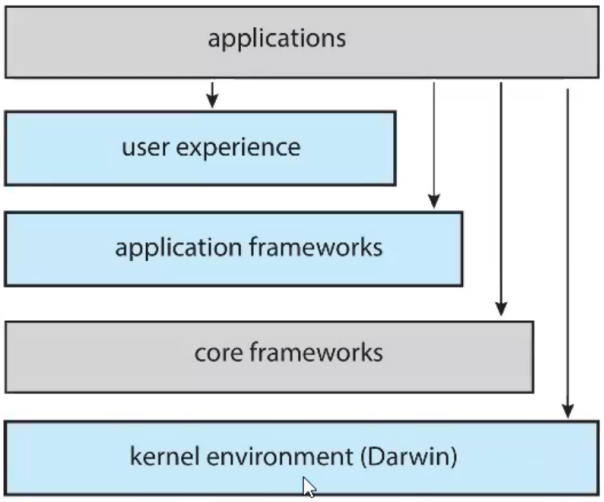
1.1 מבני מערכות הפעלה

1.1.1 מערכות היברידיות

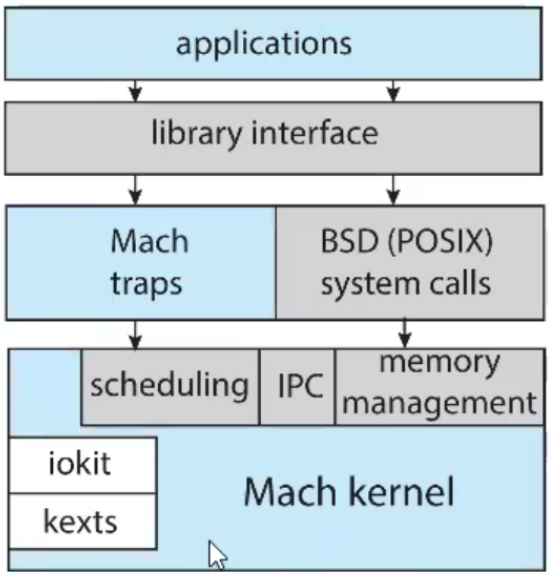
רוב המערכות המודרניות הן לא מסוג אחד טהור. מערכת היברידית משלבת גישות שונות כדי לטפל בביצועים, אבטחה וצורכי שימוש. למשל הקרנלים של לינוקס וסולאריס נמצאים במרחב הכתובות של הקרנל לכן הם מונולוטיים עם פונקציה מודולרית של העלאה דינמית. גם ווינדוס היא מערכת היברידית- ברובה היא מונוליטית, אך עם מיקרוקרנל עבור תתי-מערוכות בעלי אישיות שונה.

1.1.2 המבנה של *macOS* ו- *iOS*

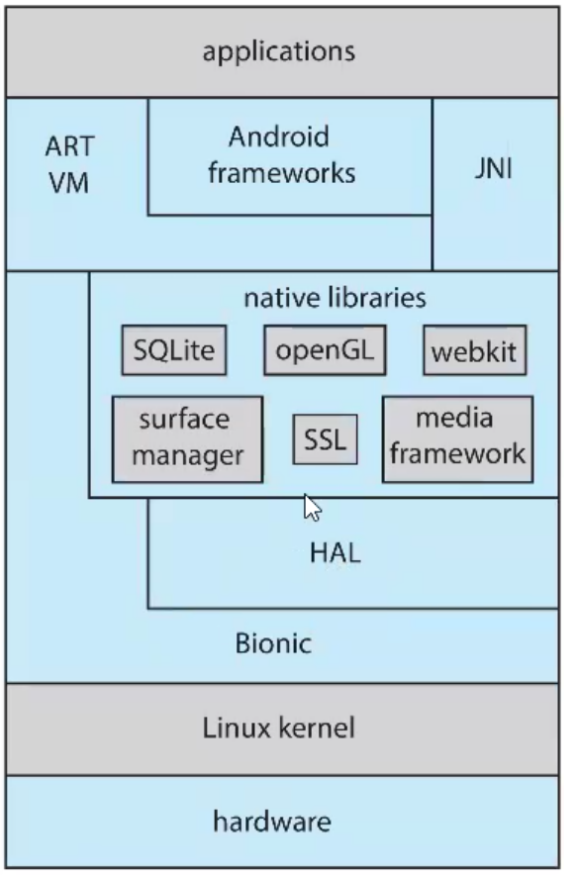
להלן תרשים המתאר את מבנה המערכות;



1.1.3 המבנה של $Darwin$



1.1.4 מבנה מערכת אנדרואיד



כעת, לאחר שלמדנו על מבנים שונים של מערכות הפעלה, נלמד איך מערכת נבנית ומאותחלת.

1.2 בניית ואתחול מערכת הפעלה

מערכות הפעלה לרוב מעוצבות לרוץ על מחלקת מערכות עם מגוון ציוד היקפי (*peripheral*) והן מותקנות על המחשב לפני שהוא נקנה. עם זאת, ניתן להתקין בהן מערכות הפעלה נוספות (למדנו כיצד לעשות זאת עם ווירטואליזציה)
לעומת זאת, אם נרצה לבנות מערכת הפעלה חדשה, נצטרך לבצע את הפעולות הבאות:

1. כתיבת ה־ *source code* של מערכת ההפעלה.
2. להרכיב את מערכת ההפעלה למערכת שעליה היא תרוץ.
3. עיבוד מערכת ההפעלה.
4. התקנת מערכת ההפעלה למכשיר.
5. אתחול (*Boot*) המחשב ומערכת ההפעלה החדשה שלה.

לדוגמה, כך ניתן לבצע שלבים אלה עם *Linux*:

1. ניתן להוריד את ה־ *source code* של *Linux*.
2. נרכיב את הקרנל דרך "*make menuconfig*"
3. נעבד את הקרנל דרך "*make*"
   - פקודה זו יוצרת את *vmlinuz*, שזהו ה־ *kernel image* (קובץ בינארי המייצג את הקרנל)
4. נקמפל את הקרנל דרך "*make modules*"
5. נתקין את המודולים של הקרנל ל־ *vmlinuz* דרך "*make modules_install*"
6. נתקין את הקרנל החדש למערכת דרך "*make install*"

כעת נתעמק יותר באתחול המערכת, כלומר ב־ *System Boot*
כאשר מתחיל זרם במערכת המחשב, מתחילה הרצה במקום מסוים בזיכרון. מערכת הפעלה חייבת להיות זמינה לחומרה על מנת שהחומרה תאתחל אותה, לכן כביכול אנחנו נמצאים בפרדוקס. אך בשביל לפתור בעיה זו, יש לנו את ה־ *bootstrap loader*.
ה־ *bootstrap loader*, המכונה גם *BIOS*, הינו חלק קטן של קוד. הוא שמור ב־ *ROM* (או *EEPROM*) שנמצא בקרנל. הוא מעלה את מערכת ההפעלה לזיכרון ומתחיל אותה. לפעמים נדרש תהליך דו־שלבי בו *boot block* הנמצא במקום מסוים מועלה על ידי הקוד של *ROM*, והוא מעלה את ה־ *bootstrap loader* מהדיסק. מערכות מודרניות החליפו את *BIOS* ב־ *Unified Extensible Firmware Interface* (*UEFI*)
BIOS שמשומש לרוב הוא *GRUB*, שמאפשר בחירה של קרנל ממספר דיסקים, גירסאות ואפשרויות קרנל. לאחר שהקרנל מועלה הוא מתחיל לרוץ. כמו כן, לרוב ה־ *BIOS* מאפשר סוגים שונים של *booting*, למשל עבור *mode single user*
עד כאן *bootstrap loader*. כעת, נעבור לאיך ניתן לדבג במערכות הפעלה.

1.3 דיבוג ושיפור מערכות הפעלה

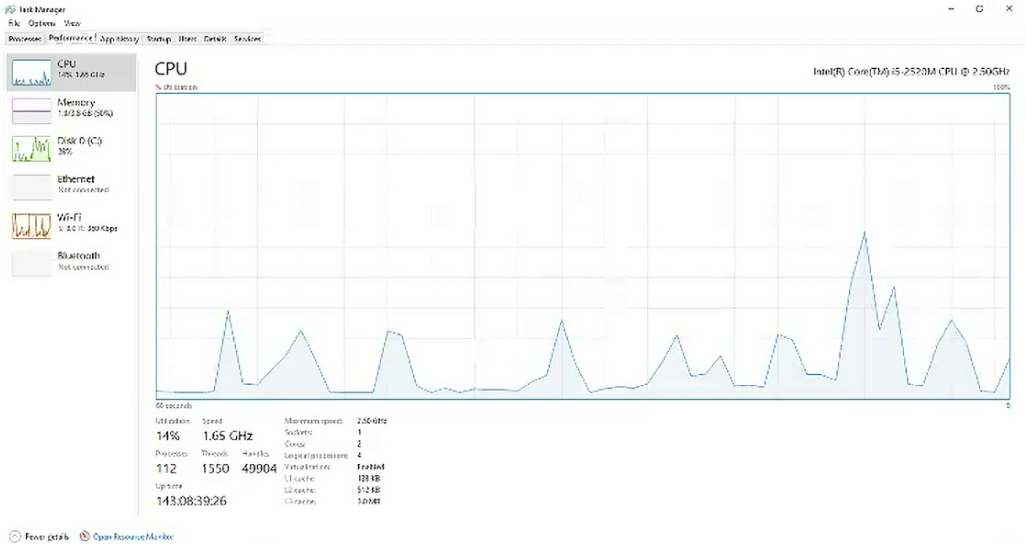
Debugging אשר מכונה גם *performance tuning*, הינה פעולה אשר מומלץ שתיתמך על ידי מערכת הפעלה, למען נוחות השימוש בה. לשם כך, מערכות הפעלה מייצרות *log files*. מעבר לכך, תקלה באפליקציה יכולה לייצר קובץ *core dump* שבו זיכרון התהליך בו היא קרתה. ניתן להשתמש במידע השמור בו על מנת לזהות את מקור הבעיה. כמו כן, תקלה במערכת ההפעלה יכולה ליצור קובץ *crash dump* עם זיכרון הקרנל.
מעבר לטיפול בקריסות, *performance tuning* יכול לשפר את ביצועי המערכת. למשל בעזרת שמירה של *listings trace* של פעילויות בשביל ניתוח. כמו כן *Profiling* דוגמיות תקופתיות יכול לסייע בזיהוי טרנדים סטטיסטיים.

לפי *Kernighan's Law*, "דיבוג קשה פי שתיים מכתיבת הקוד עצמו. לכן, כאשר אתה כותב את הקוד הכי מתוחכם שאתה יכול, לפי הגדרה אתה לא יכול לדבג אותו."

כעת נדגיש כיצד ניתן לבצע *performance tuning*.

1.3.1 *Performance Tuning*

ניתן לשפר את ביצועיה של מערכת בעזרת זיהוי והסרת צווארי בקבוק. אך לשם כך, מערכת ההפעלה חייבת לספק דרכי חישוב ותצוגה המעריכים את תפקוד המערכת. למשל, בהמשך תוכלו לראות את ה־ *Task Manager* של ווינדוס; ציינו מקודם את המושג *Traicing* המשמש ל־ *performance tuning*. זוהי פעולה שאוספת נתונים



מאירוע מסוים, כגון צעדים שנעשו עבור קריאה לקריאת מערכת. כלים אשר משמשים ל־ *tracing* כוללים בין היתר;

- *strace* ־ מסמנת קריאות מערכת אשר זומנו על ידי תהליך.
- *gdb* ־ מדבג ב־ *source level*
- *perf* ־ אוסף של כלי ביצוע של *Linux*
- *tcpdump* ־ אוסף של חבילות רשת.

אינטראקציות של דיבוג בין רמת היוזר לקוד הקרנל הן כמעט בלתי אפשריות ללא סט כלים שמבין את שניהם. *BCC* (*BPF Compiler Collection*) הוא סט כלים רחב אשר מספק שירותי *tracing* ל־ *Linux*. למשל, כאן *disksnoop.py* מסמן את פעולות ה־ *I/O* של דיסק;

TIME(s)	T	BYTES	LAT(ms)
1946.29186700	R	8	0.27
1946.33965000	R	8	0.26
1948.34585000	W	8192	0.96
1950.43251000	R	4096	0.56
1951.74121000	R	4096	0.35

כמו כן ישנם עוד כלים המשמשים את $Linux$ עבור $tracing$;