מערכות הפעלה - הרצאה 2

שרון מלטר, אתגר 17

7 ביולי 2024

1 Chapter 1 : continuation

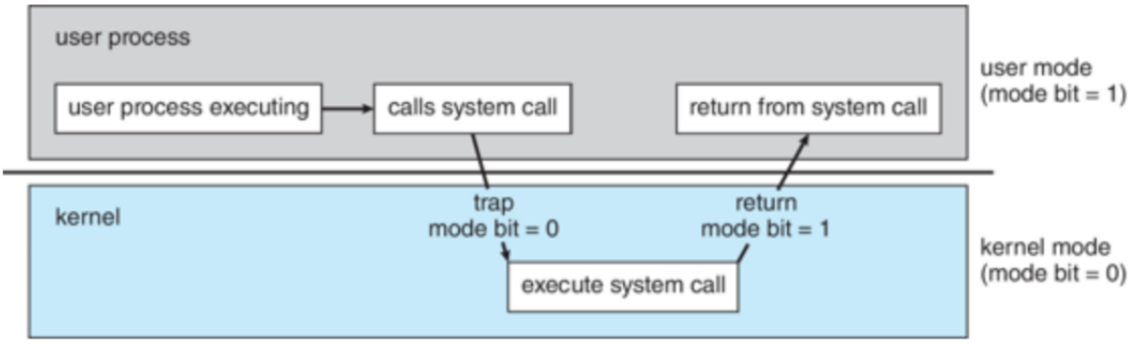
1.1 Multiprogramming + Multitasking (Timesharing)

במחשבים מודרניים לא ייתכן שמחשב יידע רק להריץ את פקודות היוזר כל פעם. הוא גם מבצע פעולות שהיוזר לא ביקש באופן אקטיבי. כמו כן, ברגע שבו משימה צריכה לחכות למשהו, מערכת ההפעלה עוברת למשימה אחרת.
אבל אף מעבד אינו יכול לבצע שתי פעולות בו זמנית. מה שאנו עושים במקום זה מולטי־טאסקינג "שקרי". מהי הכוונה לכך? הנה דוגמה;
נחשוב על עורך דין שיש לו מספר לקוחות. ביום העבודה שלו הוא יכול לכתוב מייל ללקוח א' ואז לקרוא מייל של לקוח ב', להתקשר ללקוח ג' ואז לקרוא מייל מלקוח א"... בכל רגע נתון הוא שירת לקוח אחד, אבל ביום העבודה שלו הוא טיפל בבעיות של מספר לקוחות. כלומר, מהצד ניתן לחשוב שהוא ביצע מולטי־טאסטקינג.
הרכיב שמחליט מהי המשימה שעושים כרגע הוא ה־ $Job\ \ Scheduler$, הוא בוחר בכל פעם job מתוך ה־ $memory$, ששומר תת־קבוצה של כל ה־ $jobs$ הקיימים. זאת מכיוון שהיוזר איננו יכול בעצמו לוודא שבכל רגע נתון ה־ CPU עסוק. למשל, כאשר job צריכה לחכות ל־ I/O, מערכת ההפעלה מחליפה את ה־ job שה־ CPU מבצע.
המעבד מחליף משימות מספיק מהר עד שהיוזר יכול לבצע אינטרקציה עם כל משימה בזמן שהיא רצה, מה שיוצר מחשוב אינטרקטיבי.
כמה עקרונות של $multitasking$:

- זמן התגובה צריך להיות קטן משנייה
- לכל יוזר יש לפחות תוכנית אחת שרצה בזיכרון
- אם מספר משימות מוכנות להרצה באותו הזמן, משתמשים ב־ $scheduling\ CPU$
- אם כל המשימות לא נכנסות לזיכרון, הרכיב $swapping$ מכניס ומוציא את חלקם מהזיכרון כדי שירוצו
- זיכרון וירטואלי מאפשר ביצוע של תהליכים שאינם לחלוטין בזיכרון.

1.2 Dual − Mode Operation

ברכיב החומרה של מחשב ישנו רכיב שנקרא $mode - bit$ שזהו ביט שמורה האם אנו רצים על קוד של קרנל־מוד או יוזר־מוד.
היוזר איננו יכול לשנות את הביט, אלא רק $system - call$. כלומר, רק אם היוזר קרא ל־ $system - call$ הביט משתנה, ולאחר מכן הוא מתאפס. ייתכן שנרצה לשנות את הביט כאשר אנו ביוזר־מוד מכיוון שכפי שנאמר, ישנן פעולות שניתן לבצע רק בקרנל־מוד, כגון קריאה מהדיסק.
היוזר מבקש מהקרנל שיבצע עבורו פעולות בעזרת $trap/exeption$ הינם כמו $interrupts$, אך שמקורם מהתוכנה. בעזרת $trap$ היוזר מבקש מהקרנל לבצע עבורו פעולות.
כך כוחו של היוזר מוגבל והוא לא יכול להרוס את המערכת מתוך זדון או טמטום.
להלן שרטוט של תהליך השימוש ב־ $mode - bit$; במערכות מודרניות ישנה היררכיה מורכבת יותר; ייתכן למשל $mode$ שבו ניתן לקרוא ולכתוב לדפים מסוימים וגם $mode$ בו ניתן רק לקרוא מהם, בנוסף ל־ $mode$ הנמוך ביותר בהיררכיה ול־ $kernel\ mode$



1.3 *Timer*

במערכות מודרניות ישנו *'Timer'*. תפקידו הוא לוודא שלא תרוץ לולאה אינסופית.
לכל *job* שרץ, ישנו טיימר שעוצר את ה־ *job* בעזרת *interrupt* אם הוא לא הסתיים עד שעבר זמן מוקצב ועובר ל־ *job* אחר.
אבל מה אם מדובר פשוט ב־ *job* ארוך?
לאחר שעבר הזמן המוקצב, לא זורקים את ה־ *job*. נחזור אליו לאחר *job* אחר. באופן כללי נריץ את ה־ *job* שזמנו הוא הקצר ביותר, כדי 'להיפטר ממנו'.
כפי שבוודאי הבנת, לא ניתן באמת לעצור או לזהות לופ אינסופי דרך הטיימר. כמו שלמדנו במודלים חישוביים, לא באמת ניתן לזהות לופ אינסופי עם מחשב.
קל להתבלבל, אך ה־ *program counter* הינו רכיב אחר, אשר מצביע על כתובת הפקודה הבאה שצריכה להתבצע בתהליך. באמצעותו ניתן לחזור לתהליך שנעצר בעת ביצועו.

1.4 *Process Management*

נתייחס רגע להבדל שבין *process* ל־ *program*.
תהליך, *process*, הינו תכנית (*program*) שמריצים אותה. תהליך הוא אקטיבי ותכנית היא פאסיבית. (עם זאת, תהליך עדיין צריך מעבד, זיכרון, *I/O*, תיקיות, מידע של אתחול... וכאשר מבטלים תהליך יש להחזיר את כל המשאבים שניתן להשתמש בהם שוב)
מערכת ההפעלה לא חייבת להיות עם תהליך בודד, אלא עם מספר תהליכים שהינם פריבילגיים. ככל שיש יותר תהליכים והם יותר קטנים, קל לזהות ולתקן טעויות (התהליכים יכולים לתקשר ביניהם ותקלה באחד מהם תגרום לפחות נזק)

מערכת ההפעלה אחראית על יצירת ומחיקת תהליכים של יוזרים או תהליכי מערכת,
עצירת והמשך תהליכים,
מכניקת סינכרון תהליכים,
תקשורת בין תהליכים
ומכניקה לטיפול במקרי 'מבוי סתום'.

1.5 *Memory Management*

ניהול הזיכרון מורכב מ־

- כדי לאפשר ביצוע תכנית, כל (או חלק) מהפקודות שלה צריכות להיות בזיכרון.
- בנוסף, כל (או חלק) מהדאטה שהתכנית זקוקה לה צריכה להיות בזיכרון.
- החלטה מה צריך להיות בזיכרון ומתי, על מנת למקסם את יעילות ה־ *CPU* ואת זמן התגובה של המחשב לבקשות היוזר.
- לעקוב אחרי אילו חלקים בזיכרון נמצאים בשימוש ועל ידי מי.
- לקבוע אילו תהליכים (או חלקי תהליכים) להכניס או להוציא מהזיכרון.
- להקצות ולשחרר הקצאות של מרחב זיכרון כנדרש.

1.6 $File - System\ Management$

לרוב מערכת הקבצים שמורה ב־ $hard\ disk$ ניהול מערכת הקבצים מורכבת מ־

- מערכת ההפעלה צריכה להציג את אחסון הקבצים באופן אחיד וברור. לשם כך מערכת ההפעלה תזדקק ל־
  - תכונות פיזיות מופשטות המתארות את יחידות המחסון $(file)$.
  - כל מדיום (דיסק און־קי, $hard\ drive$) מיוצג באותם תכונות. למשל, במחשב של ווינדוס אנו לא 'רואים' הבדלים בין קובץ לבין דיסק. אנו רק יודעים לשניהם יש מרחב זיכרון מסוים שהם מנצלים.
- לרוב קבצים מאורגנים ב־ $directories$.
- ניהול מי יכול לגשת לאילו קבצים.
- יצירת ומחיקת קבצים.
- מיפוי קבצים אל אחסון משני.
- יצירת שינויים בקבצים ו־ $directories$.
- ביטוח הקבצים באחסון יציב (לא נפיץ, $non - volantil$)

1.7 $Mass - Storage\ Management$

- לרוב דיסקים משומשים על מנת לשמור דאטה שלא מתאימה לזיכרון הראשי או שחייבת להישמר לפרק זמן ארוך.
- הקצאת ושחרור הקצאת זיכרון.
- ניהול מרחב זיכרון חופשי.
- ניהול לוח זמנים של הדיסק.
- חלוקת זיכרון לחלקים.
- הגנה על הזיכרון $(protection)$.

1.8 $Caching$

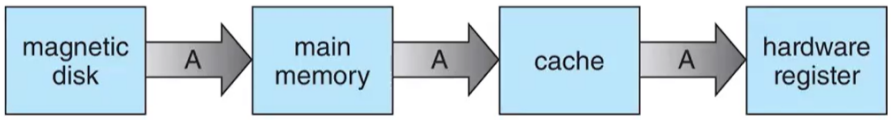
כעת נדבר על $Caching$ כפועל. הרעיון הוא שלוקחים אינפורמציה, ומעבירים אותה מאחסון איטי וגדול לאחסון קטן ומהיר יותר. כך בכל פעם שמבצעים גישה לזיכרון, קודם כל בודקים האם הדאטה הנדרשת שמורה ב־ $cache$ הקרוב־ אם כן אז ניגשים אליו ממנו ואחרת מעתיקים את הדאטה ל־ $cache$ וניגשים אליו שוב.
בגלל הבדלי הגודל, נצטרך לקבוע איזו אינפורמציה יש להעביר לאן. להלן תכונות של סוגי אחסון שונים:

Level	1	2	3	4	5
Name	registers	cache	main memory	solid-state disk	magnetic disk
Typical size	< 1 KB	< 16MB	< 64GB	< 1 TB	< 10 TB
Implementation technology	custom memory with multiple ports CMOS	on-chip or off-chip CMOS SRAM	CMOS SRAM	flash memory	magnetic disk
Access time (ns)	0.25-0.5	0.5-25	80-250	25,000-50,000	5,000,000
Bandwidth (MB/sec)	20,000-100,000	5,000-10,000	1,000-5,000	500	20-150
Managed by	compiler	hardware	operating system	operating system	operating system
Backed by	cache	main memory	disk	disk	disk or tape

Movement between levels of storage hierarchy can be explicit or implicit

1.8.1 *Cache Coherency*

סביבות בהן ישנו *multitasking* חייבות להיזהר משימוש בדאטה לא מעודכנת. למשל; אם הנתון A נשמר ב־*cache*, חשוב לעדכן אותו לאחר שמבוצע בו שינוי ברמה הנמוכה יותר של הזיכרון. פתרונות לבעיה זו נמצאים בפרק 19 בספר.

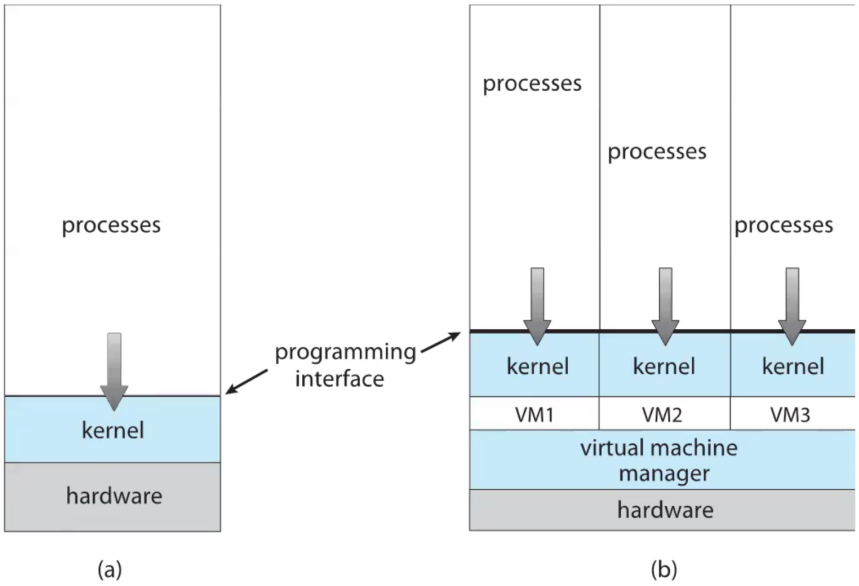


1.9 *Protection and Security*

כאשר מדברים על *protection* מדברים על הגנה מבאגים שיכולים להרוס את המערכת וכאשר מדברים על *security* מדברים על אבטחה מפני גורמים עוינים שרוצים לגנוב מידע / זהות / להרוס את המערכת.
בדרך כלל נרצה שהיררכיית המשתמשים תהיה יותר מורכבת, כלומר שלרוב היוזרים יהיו אותן פריבילגיות, אבל יהיה מספר מצומצם יותר שיוכל לקבל פריבילגיות נוספות.
היתרון בכך הוא שנוכל להוסיף פריבילגיות ליוזרים מסוימים, בלי לתת ליותר מדי מהם להשתמש באופן חופשי בקרנל־מוד. חשוב: הכוונה היא

1.10 *Virtualization*

הרעיון של וירטואליזציה עובד כך:
אם נפתח תוכנה ש'מתחזה' לחומרה, נוכל לשים בה מערכת הפעלה אחרת מהמערכת של המחשב עליו היא נמצאת. המערכת השנייה תתקשר עם התכנה המתחזה, כמו שהיא אמורה לתקשר עם חומרה. תוכנה אשר מתחזה לחומרה מכונה *VMM (Virtual Memory Manager)*.
כך ניתן להשתמש במערכות הפעלה אחרות על *Windows*. כל מה שנדרש לעשות הוא להעתיק את פקודות המערכת של מערכת אחרת. המערכת האחרת ותיחשב כ־ *guest* מכיוון שהיא לא באמת מתקשרת עם הקרנל, אלא עם *virtual machine (VM)*. להלן תרשים המתאר מערכת שבה פועלות מספר מערכות שונות; דוגמאות



לווירטואליזציה:

- לפטופ של *Apple* המריץ מערכת של *windows* בתור אורח (נהוג לומר אז שהמערכת של אפל היא 'מארחת')
- פיתוח אפליקציות למספר מערכות הפעלה ללא הפרדה לפיתוח במערכות שונות.

- בדיקת איכות אפליקציות המיועדות למערכות שונות במערכת יחידה.

שימי לב ש־ VMM יכולה גם לארח מערכת אחרת.

1.11 $Distributed \quad System$

קבוצת מערוכות שונות, ייתכן שהטרוגניות, שמרושתות ביחד.
מהי רשת? ־ רשת היא דרך תקשורת, לרוב TCP/IP. למשל;

- Local area network (LAN)
- Wide Area Network (WAN)
- Metropolitan Ara Network (MAN)
- Personal Area Network (PAN)

מערכת הפעלה רשתיות מעניקות למערכות דרך לתקשר ברשת. היא מאפשרת להן להעביר מסרים אחת לשנייה, ללא צורך בהתערבות חיצונית, ובכך משרה את האשליה שהמערכות שמתקשרות ביניהן הן אותה אחת.

1.12 $Computer - System \quad Architecture$

רוב המערכות משתמשות במעבד יחיד עם יכולות כלליות, וכמו כן במספר מעבדים עם $special - purpose$. מערכות עם מספר מעבדים צוברות פופולאריות־ הן מכונות גם־
$Tightly - Coupled\ Systems$, $Parrallel\ Systems$, $Multiprocessors\ Systems$, היתרונות במספר מעבדים הן;

- עלייה בתפוקה.
- עלייה באמינות (ישנן פחות השלכות לשגיאה שנעשתה באחד המעבדים)

ישנן שתי גישות לשימוש במספר מעבדים:

- $Asymmetric\ Multiprocessing-$ לכל מעבד ישנה משימה משלו. ישנו מעבד אשר תפקידו הוא לחלק את המשימות בין שאר המעבדים.
- $Symmetric\ Multiprocessing-$ כל מעבד יכול לבצע כל משימה. זוהי הגישה בה רוב המערכות נוקטות.

כיצד ניתן לממש מולטי־מעבדים סימטריים:

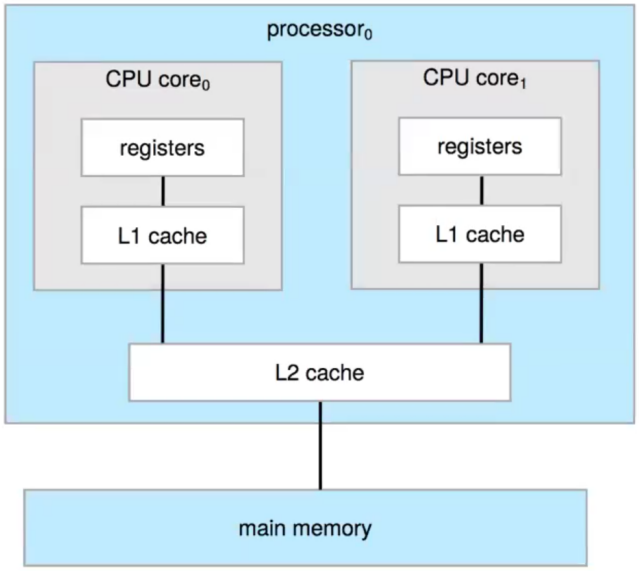
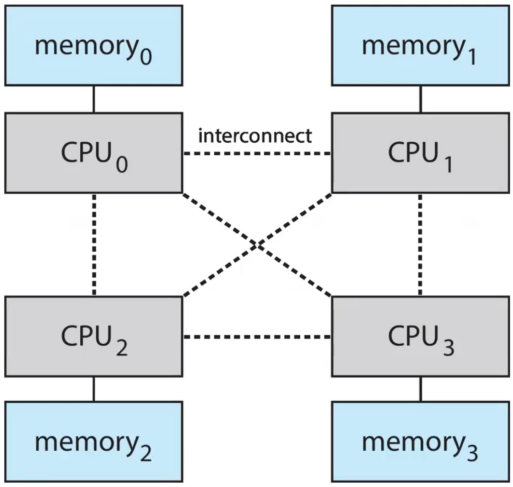


Figure 1: Dual-Core multiprocessors system

1.12.1 $NUMA$

במערכות עם מספר מעבדים, יכול להיות מצב שבו לכל אחד מהם יש את רכיב ה־ $memory$ הקרוב אליו וכל אחד מהם יכול לקרוא את רכיבי ה־ $memory$ של האחרים, הם קוראים באחד הקרוב אליהם במהירות מירבית ולכן הם משתמשים בו. למבנה זה קוראים $Non-Uniform\ Memory\ Access\ Systems\ (NUMA)$. להלן תרשים המתאר את המנגנון:



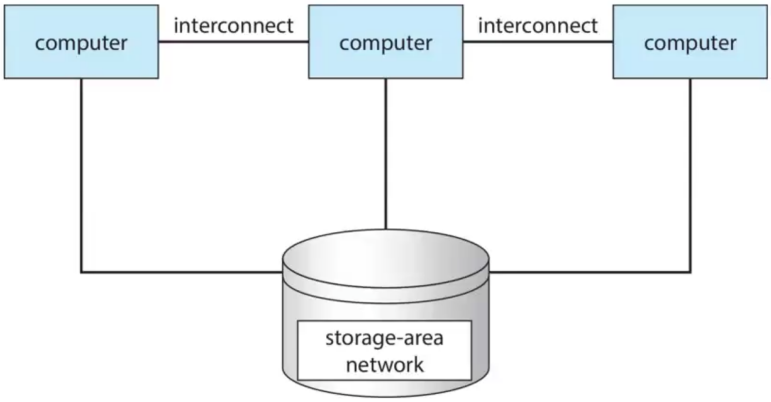
1.12.2 $Clustered\ Systems$

כמו מספר מעבדים שעובדים ביחד, $clustered\ systems$ אלו קבוצת מערכות שעובדות ביחד. ישנה מערכת כללית ומספר מערכות עם מטרות יותר מסוימות. לרוב, הן חולקות זיכרון באמצעות $storage-area\ network\ (SAN)$. הן מספקות $services$ אמינים יותר, עקב אמידות גבוהה יותר לתקלה באחת המערכות (או אפילו אי־תפקוד של אחד המחשבים)
ישנן שתי גישות למימוש מערכות מקובצות:

1. $Asymmetric\ clustering-$ ישנה מערכת אחת הנמצאת במצב $hot-standby$. במצב זה, אם ישנה תקלה או ששאר המערכות צריכות שתבוצע פעולה נוספת, אותה מערכת יכולה לפעול כאשר קוראים לה. כמו כן, היא מעודכנת על המתרחש בשאר המערכות (לפי רמת הצורך) כך שאין צורך לחכות כאשר קוראים לה לביצוע משימה.

2. $Symmetric\ clustering-$ ישנם מספר רכיבים אשר מריצים אפליקציות ומפקחים אחד על השני.

להלן תרשים לשיתוף נתונים במערכות מקובצות:



1.13 Computing Environments

ישנם מספר מקומות שדרכם ניתן לבצע חישובים.

1.13.1 Traditional

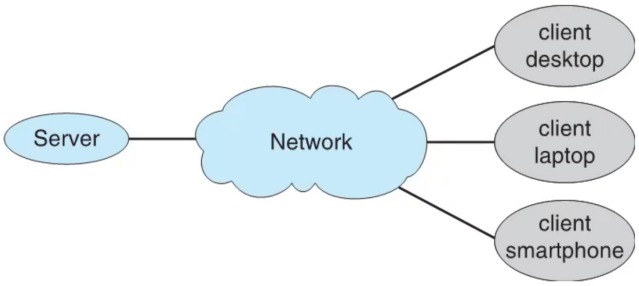
אלו מכונות חישוב שעובדות כשלעצמן והן $general-purpose$.

1.13.2 Mobile

מכשירים כמו טלפון, אייפד...
ההבדל בינם לבין $traditional$ הוא של־ $Mobile$ יש יותר $features$ OS כגון $GPS,$ $gyroscope$ וניתן להשתמש באפליקציות מסוג חדש כגון מציאות רבודה.
משתמשים בהם ברשתות $wireless$ או סלולאריות בשביל תקשורת. בשביל חיבור, מערכות אלה משתמשות בנתונים סלולאריים או בחיבור ללא כבל.

1.13.3 Client Server

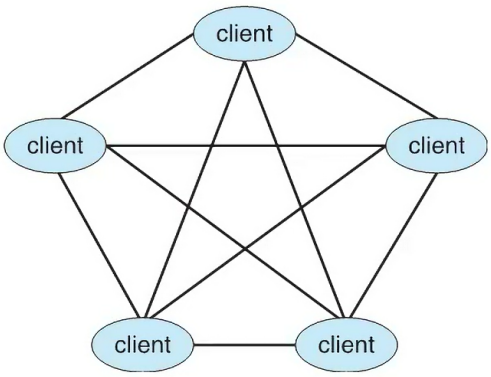
עבור חישובים אלו, טרמינלים פרימיטיביים הוחלפו במחשבים אישיים חכמים. מערכות רבות עכשיו הן $servers$ ומגיבות לבקשות שיוצרים לקוחות. במילים פשוטות, דרך זו של חישוב היא מסר של בקשת חישוב הנוצרת במערכת של יוזר, ביצוע החישוב במערכת $server$ וקישור ביניהן באמצעות $Network$. להלן תרשים המייצג זאת;
ישנן שני סוגים של מערכות $Client-Server$;



1. $Compute-server$ $System-$ מערכת זו מספקת ממשק אשר מבצע שירותים (נקראת גם ממסד נתונים)
2. $File-server$ $System-$ מערכת זו מספקת ממשק לאחסון ושליפת קבצים.

1.13.4 Peer-to-Peer

זהו מודל נוסף של $distributed$ $system$, בו אין הבדל בין לקוח לשרת־ הם שניהם נחשבים $'peers'$ ויכולים לתפקד או כשרת או כלקוח.
כל $peer$ נרשם בעזרת $central$ $lookup$ $service$ או שכל $peer$ מבקש שירות או עונה לבקשת שירות בעזרת $discovery$ $protocol$. להלן תרשים לדוגמה;

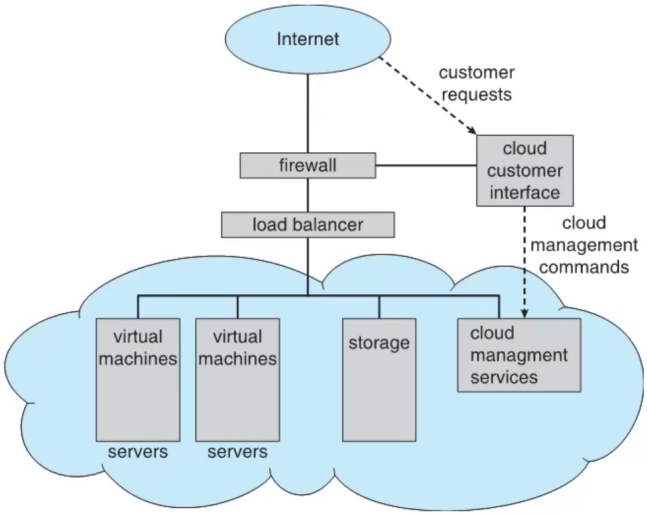


1.13.5 $Cloud\ \ Computing$

סוג זה מאפשר חישוב, אחסון ואפילו אפליקציות כ־ $service$ בין רשת. ווירטואליזציה משומשת בתור בסיס אך ישנה גם תוספת לוגית. למשל, חוות המחשבים של גוגל אשר מאפשרות לאנשים לשמור זיכרון רב בענן. יש מספר סוגים של $Cloud\ Computing$;

1. ענן ציבורי ־ זמין לכולם דרך האינטרנט.
2. ענן פרטי ־ משומש על ידי חברה פרטית בשביל מטרותיה.
3. ענן היברידי ־ כולל רכיבי ענן ציבורי ופרטי.
4. תוכנה כ־ $service\ SaaS$ כלומר, מספר או יותר אפליקציות ניתנות לגישה דרך האינטרנט.
5. $Software\ as\ a\ Service\ (SaaS)$־ מחסנית של תוכנות המוכנות לאפליקציה דרך האינטרנט. $(server\ database\ i.e)$
6. $Infrastructure\ as\ a\ Service\ (IaaS)$־ שרת או אחסון שניתן לגשת אליו דרך האינטרנט $(for\ backup\ use\ storage\ avilable\ i.e)$

סביבת חישוב ענן מורכב ממערכות הפעלה מסורתיות, $VMMs$ וכלים לניהול הענן. כמו כן, מאחר שנדרשת גישה לאינטרנט, $firewall$ משמש להגנה על דאטה. תרשים לתיאור חישוב דרך ענן;



1.13.6 $Real-Time\ \ Embedded\ \ Systems$

כאן אם פעולה נמשכת זמן רב מדי, אין טעם למערכת.
למשל, אם ברקס במכונית יפעל לאט מדי אז המכונית כולה חסרת תועלת.

1.14 $Free\ \ and\ \ Open-Source\ \ Operating\ \ Systems$

אלו מערכות שניתן לגשת לקוד שלהן, למשל אפליקציות אנדרואיד ומערכת $Unix$. לעומת $Windows$ ו־ IOS, שלא ניתן לראות את הקוד שלהן.
שימי לב;
הכוונה ב־ $open-source$ היא שיוזרים יכולים לשנות את הקוד והכוונה ב־ $free$ היא שניתן לראות אותו. העיקרון הנגדי לעיקרון $free\ open-source$ הוא $copy\ protection$ ו־ $Digital\ Rights\ Management\ (DRM)$, לפי העיקרון הנגדי יש לשמור על זכויות היוצרים של כותבי הקוד של מערכות ההפעלה ובכך לבטח שיוכלו להמשיך להתפרנס ממנו.

1.15 $Kernel\ \ Data\ \ Structure$

שרשראות, עצי חיפוש, טבלת גיבוב.
את מכירה :)
כעת נעבור לפרק השני של הספר.

2 Chapter 2: Operating – System Services

2.1 Chapter Outline

- Operating System Services
- User and Operating System-Interface
- System Calls
- System Services
- Linkers and Loaders
- Why Applications are Operating System Specific
- Design and Implementation
- Operating System Structure
- Building and Booting an Operating System
- Operating System Debugging

2.2 Objectives

בפרק זה נלמד;

1. לזהות שירותים המסופקים על ידי מערכת הפעלה.
2. לתאר איך *system calls* משומשות כדי לספק *services* הניתנים למערכות הפעלה.
3. להשוות ולמצוא הבדלים בין אסטרטגיות מונוליטיות, שכבתיות, מיקרוקרנל, מודלריות והיברידיות לעיצוב מערכות הפעלה.
4. לתאר את התהליך של אתחול מערכת הפעלה.
5. ליישם כלים לפיקוח על תפקוד מערכת הפעלה.
6. לעצב וליישם מודולים לקרנל לתקשורת עם קרנל של *Linux*.

2.3 Operating System Services

מערכות הפעלה מספקות סביבה להרצת תוכניות ושירותים בשביל המשתמש. להלן מספר פונקציות שמועילות ליוזר;

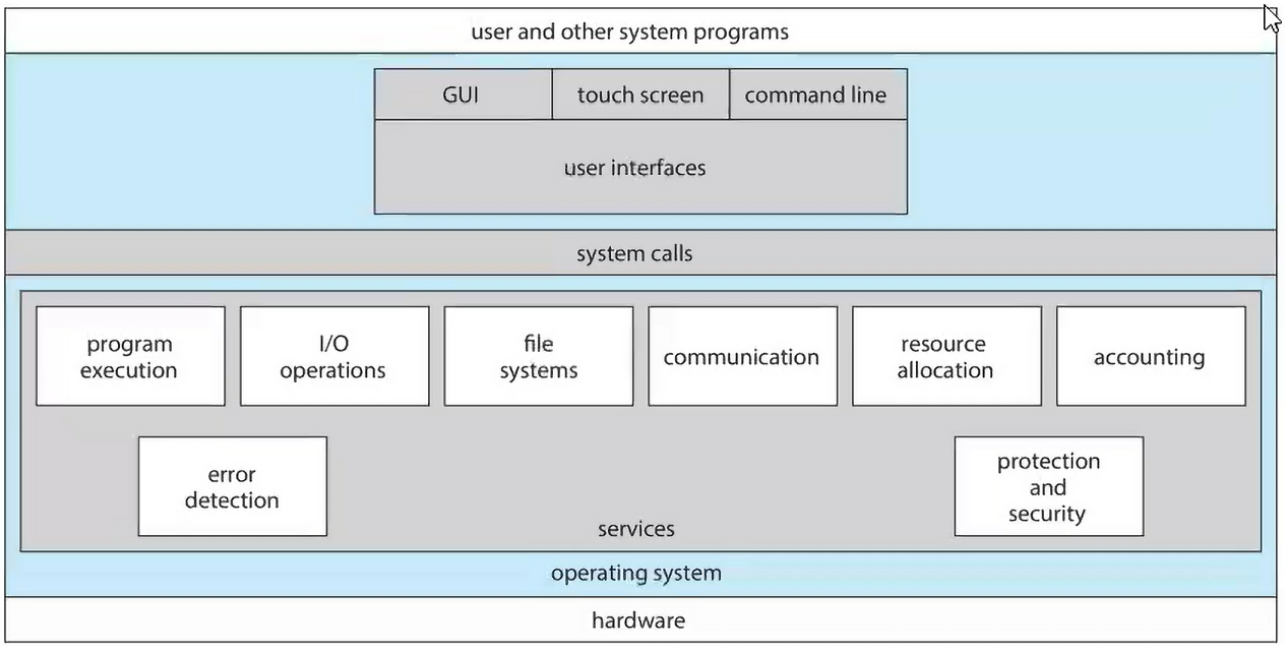
- ממשק משתמש (UI) - נמצא בכמעט כל מערכת הפעלה. דוגמאות: מסך מגע, $Command-Line\ (CLI)$ וממשק גרפי GUI.
- הרצת תוכנית - המערכת חייב להיות מסוגלת להעלות תוכנית לזיכרון, להריץ אותה ולסיים את ההרצה באופן נורמלי או לא נורמלי (מה שמרמז על שגיאה)
- פעולות I/O - תוכנית שרצה יכולה להזדקק ל- I/O, לכן ייתכן שיידרש קובץ או התקן I/O (עכבר, מקלדת...)
- שימוש במערכת קבצים - תוכניות צריכות לקרוא ולכתוב למערכת הקבצים ול- *directories*, כמו כן ליצור ולמחוק אותם ולחפש ביניהם. מכאן נדרש גם ניהול גישה לקבצים השונים.
- תקשורת - ייתכן שתהליכים יזדקקו להחליף ביניהם מידע, באותו המחשב או בין שני מחשבים שונים ברשת. התקשורת יכולה להתבטא בשימוש במידע משותף או בהעברת מסרים.

- זיהוי שגיאות - מערכת ההפעלה צריכה להיות בידיעה מתמדת על שגיאות אפשריות. ייתכן שיהיו תקלות ב־ CPU ובחומרת הזיכרון, בהתקני I/O ובתוכנת היוזר. לכל סוג של שגיאה, מערכת ההפעלה צריכה לפעול בהתאם על מנת לאפשר חישובים נכונים והמשכיים. כמו כן, כלי דיבאגינג שהמערכת נותנת יכולים לייעל את יכולות היוזר להשתמש בה.
-

סט נוסף של פונקציות קיים בשביל להבטיח פעילות יעילה של המערכת עצמה דרך שיתוף משאבים:

- הקצאת משאבים - כאשר מספר יוזרים או מספר $jobs$ הרצים באופן מקביל, המשאבים צריכים להתחלק בין כולם. המשאבים הם מסוגים שונים, וכוללים מעגלי CPU, זיכרון ראשי, אחסון קבצים והתקני I/O.
- $Logging$ - לעקוב אחרי אילו יוזרים משתמשים באילו וכמה משאבים מהמחשב.
- הגנה ואבטחה - ייתכן כי בעלי המידע השמור במערכת פרטית או ברשת יירצו לשלוט בו. לכן תהליכים מקבילים לא צריכים להתערב אחד בשני. **הגנה** היא ווידוא שכל הגישות למשאבי המערכת נשלטים. למשל, שהמערכת מגיבה בהתאם לבקשה לקרוא את הערך שבתא -3 במערך. **אבטחה** של מערכת דורשת זיהוי המשתמשים, מה שמורחב להגנת התקני I/O חיצוניים מניסיון גישה בלתי חוקי.

לסיכום, נראה תרשים המציג את שירותיה של מערכת הפעלה;



2.4 $User\quad Interfaces$

2.4.1 $Command\quad adLine\quad Interpeter\quad (CLI)$

בעזרת CLI היוזר יכול לתת פקודות ישירות, שלא נעשות דרך ממשק. לפעמים ה־ CLI מיושם בקרנל ולפעמים על ידי תוכנית המערכת ולפעמים מיושם ב־ $shell$, זהו המקרה כאשר ישנם מספר $flavors$ אשר ממשים את ה־ CLI. ($shell$ הינו הממשק של ה־ CLI), $'flavors'$ שמכונים גם $linux\quad distors$ הינם מקבצים שונים של חבילות תוכנה, סביבות שולחן עבודה והגדרות מערכת.
לפעמים הפקודות שמשתמשים בהן הן $built-in$ (כלומר הן חלק מ־ $shell$) ולפעמים אלו רק שמות של תוכניות קיימות.
אם מדובר במקרה השני, אז הוספת אפיונים נוספים לא דורשת שינוי של ה־ $shell$.
יתרונות של CLI: כאשר מכירים את הפקודות, העבודה במחשב מהירה ונוחה יותר. הפקודה $mun+*command*$ מסבירה לנו מה פקודה עושה.
נחשוב על דוגמה של פתרון $wordle$ עם CLI; ניתן בעזרת הפקודה $more$ לראות את כל הטקסט שיש בקובץ. בעזרת $grep$ נבחר רק את המילים שיש בהם למשל את האות H כאות שנייה ואין בהם את האותיות $k, c, ...$ והן באורך 5. (יש את הפקודות לכך בהקלטה) וכעת יש לנו רשימה של כל המילים אפשריות לפתרון.

2.4.2 $User\ \ Operation\ \ System\ \ Interface\ \ -GUI$

זהו $metaphor\ \ interface$, כלומר סט של כלים ויזואליים, פעולות ופרוצדורות שלא מראות ליוזר מידע שהוא כבר ראה מחלק אחר של הממש. מטרתה־ $metaphor\ \ interface$ היא לתת ליוזר לא את כל המידע, אלא רק מידע מזדמן.
ממשק GUI מבוסס על גרפיקה, אייקונים, תפריטים ושימוש בעכבר לאינטרקציה עם המערכת.
רוב המערכות כיום מכילות גם ממשק CLI וגם ממשק GUI;

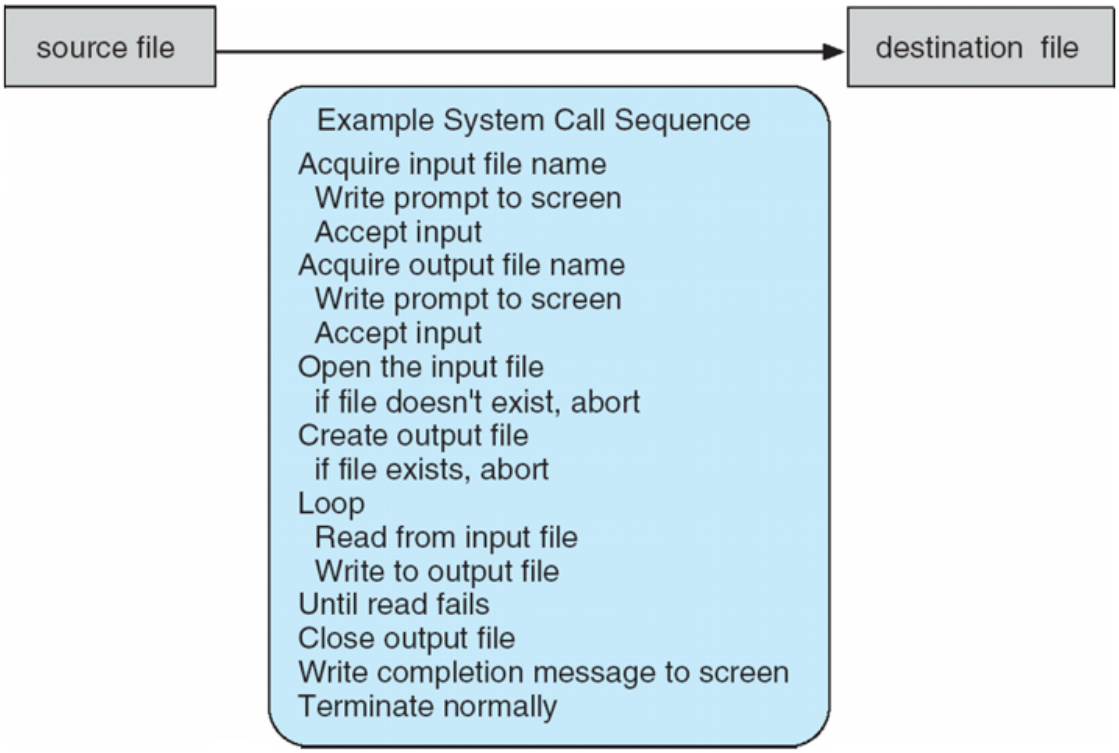
- המערכת של $Windows$ היא GUI עם $shell$ $"command"$ CLI.
- ב־ $Apple\ \ Mac\ \ OS\ \ X$ יש ממשק GUI עם קרנל של $UNIX$ מתחתיו ואפשרות להשתמש ב־ $shells$ שונים.
- ל־ $Unix$ ו־ $Linux$ יש ממשקי GUI אופציונאליים $(CDE,\ \ KDE,\ \ GNOME)$

2.4.3 $Touchscreen\ \ Interfaces$

כאשר מכשירי $Touchscreen$ פותחו, הם נזקקו לממשק חדש;
לא ניתן ולא רצוי להשתמש בעכבר, פעולות מבוססות על תנועות ידיים ועל פקודות קול וישנה מקלדת וירטואלית.

2.5 $System\ \ Calls$

זהו ממשק תכנותי ל־ $service$ שניתן על ידי מערכת ההפעלה, ולרוב נכתב ב־ $C, C++$. בדרך כלל הוא ניתן על ידי שפות עילית. לרוב הגישה נעשית דרך ממשקי אפליקציות עילית ולא על ידי $system\ \ calls$ ישירות.
שלושת הממשקים הכי פופולאריים היום הם $Win32$ של $Windows$, $Posix\ \ API$ ו־ $Linux$.
להלן דוגמה של $systemcall$;



וכמו כן הנה דוגמה ל־ $API\ (Application\ Program\ Interface)$ סטנדרטי עבור פקודה, במקרה זה הפקודה $read()$;

EXAMPLE OF STANDARD API

As an example of a standard API, consider the `read()` function that is available in UNIX and Linux systems. The API for this function is obtained from the `man` page by invoking the command

```
man read
```

on the command line. A description of this API appears below:

```
#include <unistd.h>

ssize_t     read(int fd, void *buf, size_t count)
```

return value — function name — parameters

A program that uses the `read()` function must include the `unistd.h` header file, as this file defines the `ssize_t` and `size_t` data types (among other things). The parameters passed to `read()` are as follows:

- `int fd`—the file descriptor to be read
- `void *buf`—a buffer into which the data will be read
- `size_t count`—the maximum number of bytes to be read into the buffer

On a successful read, the number of bytes read is returned. A return value of 0 indicates end of file. If an error occurs, `read()` returns −1.

כך ניתן להשתמש ב־ API של הפקודה $read$ ב־ $Linux$.