מערכות הפעלה - הרצאה 1

שרון מלטר, אתגר 17

7 ביולי 2024

1 הקדמה

כמעט כל קורס של מערכות הפעלה, מלמד לפי "הספר עם הדינוזאורים", הלא הוא $operating\ \ systems\ \ concepts$.
הספר מכיל גם גירסאת שקפים, ואלו המצגות היחידות של הקורס. (החומר של הקורס מוכל בספר)
בספר יש את החומר התאורטי, כולל כל מה שנדבר עליו בהרצאות וגם תרגילי קוד (שכתובים ב- PDF)
חומר הספר נחשב קריטי ובסיסי עד כדי כך שכמעט כל ראיון עבודה יכלול שאלות עליו.
בגדול, אנחנו נלמד כיצד מחשבים פועלים.
נלמד על תיאום בין רכיבים שונים, על כיצד מערכות קיימות פועלות ועל האם אופן בנייתן נעשה מבחירה מושכלת או במקרה.
נלמד את פרקים $1-11$ מהספר, או מעט פחות אם לא נספיק, אך עדיין מומלץ לקרוא את שאר פרקי הספר.
אסור להעתיק מתלמידים אחרים, מתרגילי בית של סטודנטים משנים קודמות ומהאינטרנט. אם לקחת קוד מהאינטרנט, יש לדווח לרחל. עם זאת מותר ללמוד גם מכל מקור אינטרנטי, פשוט לא להעתיק ממנו ללא אישור.

2 מבוא

במבוא נעשה סקירה של הכל, לרוב ב- $high-level$ אבל גם עם נקודות עם רמת פירוט גבוהה יותר.
נדבר על איך עובדת מערכת מחשב, על רכיביה, כגון מעבד, זיכרון, וכו', ועל איך מערכת ההפעלה מנהלת את כולם. נלמד על איך נגן על המערכת מפגעים כמו וירוסים או אדם שעשה טעות וכתב קוד שפוגע במחשב. נלמד גם על מערכות וירטואליות, ששולטות על מחשבים מרחוק ועוד.

תחילה נדבר על איך מארגנים מערכת הפעלה ועל איך מתקשרים בין הרכיבים השונים ועל ההפרדה הבסיסית והקריטית בין היוזר-מוד לקרנל-מוד. לאחר מכן נדבר על מערכות הפעלה בחיים האמיתיים ולמה הם משמשות.
לבסוף נדבר על $open-sources$.

מהי מערכת הפעלה?
מערכת ההפעלה של המחשב של רובנו היא ווינדוס, ומערכות ההפעלה של הטלפונים שלנו הן אנדרואיד או ios.
כמו כן במכוניות של היום יש מערכות הפעלה. למעשה, לכל מכשיר עם מחשוב יש מערכת הפעלה, גם במטוסים, טוסטרים ומכונות כביסה.
נגדיר מערכת הפעלה כך: תוכנה שמקשרת בין האדם למכונה. אין הגדרה מוסכמת, אך בקורס זה נתעסק יותר בהנדסה ופחות בהיבט המשפטי.
אפשר לחשוב על מערכת הפעלה גם בתור התכנית שרצה במחשב כל הזמן, או מה שאפשר לקנות כאשר מחפשים מערכת הפעלה...
כמובן שנרצה בעקבות כך שמערכת הפעלה תהיה נוחה לשימוש. בווינדוס למשל, הממשק נוח מאוד, אך יש בו את פונקציית cmd, שהיא מערכת מסוג $command-line$ שבה אנו צריכים לכתוב את הפקודות למערכת ההפעלה בעצמנו.

ניתן לפרק מערכת מחשב ל-4 חלקים:

1. חומרה - $CPU,\ \ Memory,\ \ I/O\ \ devices$
2. מערכת הפעלה- שכאמור שולטת ומתאמת שימוש בחומרה בין תוכניות ואפליקציות שונות.
3. אפליקציות תוכנה- מגדירה את הדרכים שבהן המערכת משתמשת במשאבים כדי לפתור בעיות חישוב של היוזרים. למשל משחקי מחשב, מעבדים להרצת קוד, מערכות דאטה...

4. יוזרים־ אנשים, מכונות ומחשבים אחרים.

ברור שנרצה שאפליקציות יהיו נוחות ביותר לשימוש, לכן במערכות מחשוב שמיועדות לקבל הרחב היוזר מתקשר באופן לא ישיר עם מערכת ההפעלה.
כאשר ישנם מספר יוזרים לאותה מערכת ההפעלה, למשל כאשר ישנו מחשב חזק שמספר עובדים משתמשים בו, נרצה שהמשאבים יחולקו בין כולם. לכן מערכות הפעלה משמשות גם כמקצות משאבים ומנהלות ביצועי בקשות יוזרים ביעילות. לולא ניהול זה ייתכן כי יוזר אחד 'ישתלט' על מרבית המשאבים.

2.1 $Interrupts$

אלו רכיבים אשר מודיעים כאשר הסתיים בוצעה פעולה רלוונטית, ומכריזים שכעת יש לעשות לטפל במשהו בעקבותיה או בעקבות התוצאות שלה. בעקבות זאת המחשב מפסיק את מה שהוא ביצע כרגע. כדי לחזור אליו, שומרים את המצב $(state)$ שהיינו בו כרגע.
ה־ $interrupts$ שמורים בווקטור, כך שאם ה־ $interrupt$ ה־2 הוא האחד שנקרא, ניגש לווקטור ונשיג משם את הפקודות שאומרות מה לעשות כדי 'לפטל' במה שדרוש. שימי לב, יש וקטור $interrupts$ יחיד למערכת. זהו חלק ממרחב הזיכרון של ה־ $kernel$.
באופן דומה, כאשר זורקים $exeption$ עוצרים את פעילות התוכנית/קוד ואולי מטפלים בו דרך $catch$. המנגנון השומר את מצב ה־ CPU (כלומר שומר את מצב הרגיסטרים וה־ $program\ counter$) ולאחר מכן מבצע את הפעולות הנדרשות לטיפול באותו $Interrupt$ נקרא $Handling\ Interrupt$

2.1.1 $Interrupt\ \ Handling$

מערכת ההפעלה שומרת את מצב ה־ CPU על ידי שמירת הרגיסטרים וה־ $counter$ של התכנית. מדובר רק בשמירה של כתובת ולכן אין הכבדה על הזיכרון.

2.2 $I/O\ \ Structure$

ישנן שתי דרכים לטפל ב־ I/O;

1. אחרי שפעולת ה־ I/O מתחילה, המערכת חוזרת לתכנית היוזר רק אחרי סיומה.
כמובן שכך ניתן רק לבצע פעולת I/O אחת בכל פעם ול־ CPU מועברת פקודה לחכות עד ה־ $interrupt$ הבא.
2. אחרי שה־ I/O מתחיל, המערכת חוזרת לתכנית היוזר ולא מחכה לסיום של I/O.
$system-call$; בקשה למערכת ההפעלה לאפשר ליוזר לחכות לסיום של I/O.
$Device-status\ \ table$; טבלה שמכילה כניסה לכל התקן I/O שבה הסוג, כתובת ומצב. המצב מתאר האם ההתקן חופשי, מבצע פעולה, או סיים לבצע פעולה. באמצעות מצב זה אנו יודעים מתי הסתתימה פעולת ה־ I/O וניתן לחזור לפעולה שביצע המעבד לפני $interrupt$.

2.3 $Storage$

2.3.1 $Storage\ \ Structure$

מבנה המחסן הוא:

- זיכרון מרכזי ־ פועל לפי $(DRAM)\ \ random-access\ \ memory$, בדרך כלל גם $volatile$, כלומר הוא 'נפיץ', ובהקשר שלנו הכוונה היא שהוא נעלם ברגע שבו מכבים את המחשב.
- זיכרון משני־ הרחבה של הזיכרון הראשי שנותן קיבולת אחסון לא נפיצה רבה.
- $Hard\ \ Disk\ \ Drives\ \ (HDD)$־ פלטת מתכת או זכוכית שמכוסה בחומר מגנטי.
משטח הדיסק מחולק למסלולים $(tracks)$, והם בתורם מחולקים לענפים $sectos$.
בחלק זה נמצא גם ה־ $diskcontroller$ שקובע את האינטרקציה הלוגית בין ההתקן והמחשב.
- התקני זיכרון לא נפיץ $(Non-volatile\ \ memory\ \ (NVM))$ הם מהירים יותר מ־ $hard\ \ disk$ ואינם נפיצים (אינם יכולים 'להיעלם' בנוחות)

2.3.2 *Storage Definitions and Notations*

יחידת בסיס הזיכרון של כל מחשב היא ביט, שיכול להיות 0 או 1. לאחר מכן ישנו בייט, שבו 8 ביטים. המקום היחיד בו סופרים לפי ביטים ולא לפי בייטים הוא רשתות תקשורת, מכיוון שרשתות תקשורת מעבירות ביט אחד של דאטה בכל פעם. 'מילה' היא מושג פחות מוגדר, שמתארת את יחידת הדאטה הבסיסית של ארכיטקטורת מחשב כלשהי, שיכולה להיות בייט אחד או יותר. למשל, מחשב עם רגיסטרים של $64-bit$ וכתובת זיכרון של $64-bit$ הוא בדרך כלל בעל מילה של 8 בייטים. מחשב יבצע את פעולותיו לפי גודל המילה שלו ולאו דווקא לפי בייט יחיד.

אחסון מחשב, כמו רוב המחשב כולו, בדרך כלל מנוהל לפי בייטים וקבוצות בייטים;

(KB) *Kilobyte*־ אלו $1,024$ בייטים.

(MB) *megabyte*־ אלו $1,024^2$ בייטים.

(GB) *gigabyte*־ אלו $1,024^3$ בייטים.

(TB) *terabyte*־ אלו $1,024^4$ בייטים.

(PB) *petabyte*־ אלו $1,024^5$ בייטים.

מדי פעם מעגלים וקוראים ל־ MB מיליון ביטים ול־ GB ביליון ביטים.

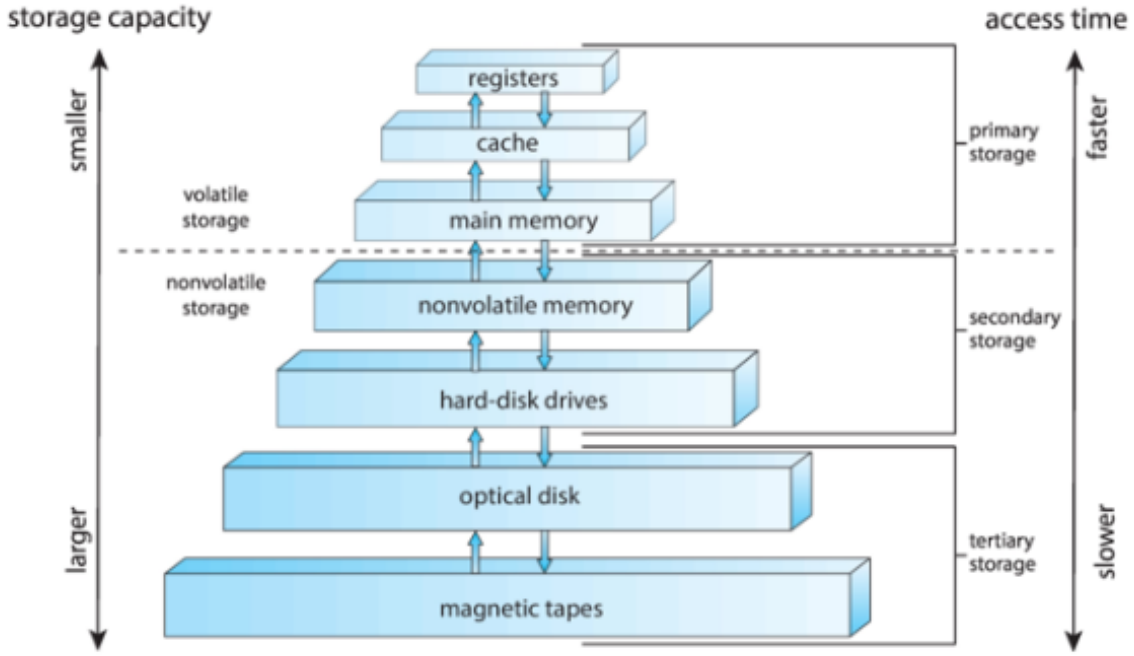
2.3.3 *Storage Hierarchy*

מערכות האחסון מאורגות בהיררכיה, כך שככל ההיררכיה כך עולה מהירות, נפיצות ומחיר הזיכרון. לכן פחות מידע מאוחסן ברמות הגבוהות יותר.

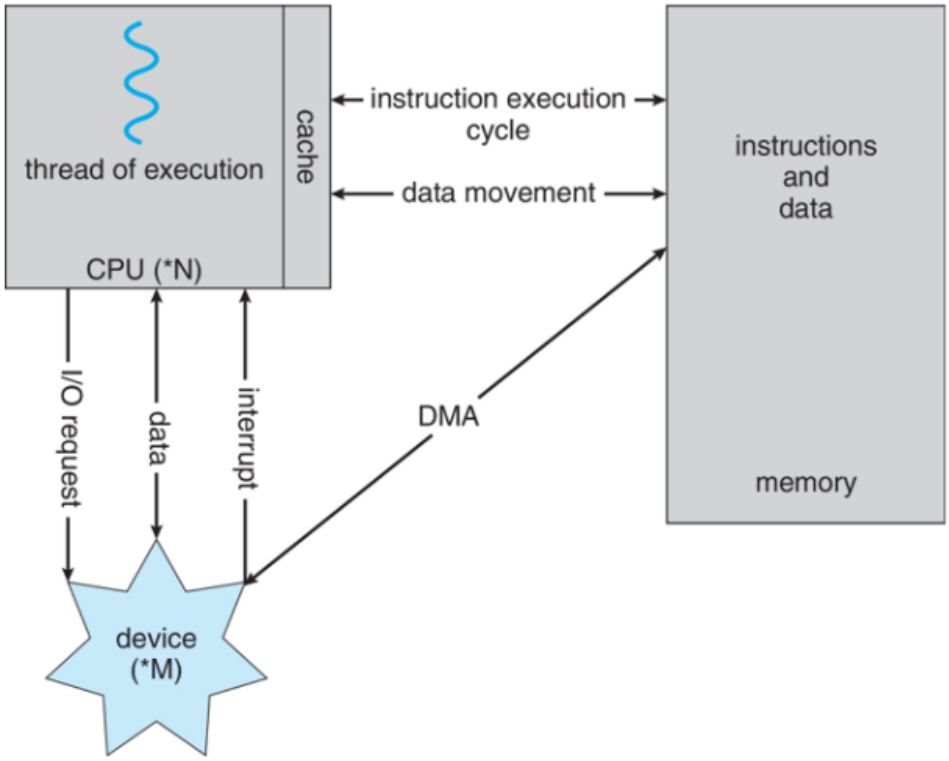
הפעולה *caching* היא העתקת דאטה למערכת אחסון מהירה יותר. למשל, זיכרון ראשי יכול להיחשב כ־ *cache* עבור הזיכרון המשני.

לכל *device controller* ישנו *Device Driver* שמנהל את I/O (פעולות $Input/Output$). הוא מהווה ממשק אחיד בין ה־ *controller* לקרנל.

היררכיית הזיכרון המלאה; כמובן שככל שמהיר יותר להגיע לזיכרון, מחירו עולה. לכן כמות הזיכרון שניתן



לשמור בו קטנה.



A von Neumann architecture

2.3.4 $Direct\ \ Memory\ \ Access\ \ Structure$

זהו מבנה שמשומש עבור התקני I/O שמהירותם קרובה למהירות הזיכרון.
במבנה זה ה־ $device\ \ controller$ מעביר בלוקים של דאטה מאחסון הבאפר ישירות לזיכרון הראשי, ללא התערבות של ה־ CPU. כמובן שבמבנה זה יש להיות זהירים יותר עם פקודות I/O.
רק $interrupt$ יחיד מופק עבור כל בלוק, במקום אחד עבור כל בייט. בשרטוט למעלה מתוארת מערכת אשר משתמשת גם ב־ DMA, בה בלוקים עוברים בין הזיכרון הראשי $(memory)$ לבין התקני $I/O\ (device)$.

2.4 כמה נקודות ספציפיות

2.4.1 $Operatin-System\ \ Operations$

כאשר אנו מדליקים את המחשב, הדבר הראשון שקורה הוא שמריצים את תכנית ה־ $bootsrap\ \ program$. מה הכוונה?
אנו רוצים להפעיל את המחשב, אך עוד אין מחשב שיקבל פקודות ויבצע אותן. אז מה עושים?
ה־ $bootsrap\ \ program$ מתחיל כאשר עובר חשמל במחשב. הוא מופעל באמצעות חומרה. הוא קוד פשוט שמתאחל את המערכת וטוען את הקרנל.
אותה תכנית מתחילה־ $system\ \ daemons$ (כן, כמו שדים) שאלו $services$ (קודים שמפיקים פונקציות ספציפיות לאפליקציות או ל־ $services$ אחרים שמשמשים את המערכת)
בזמן האתחול, הקרנל עובד לפי $interrupt\ \ driven$ שזהו מצב בו הקרנל איננו מחפש באופן אקטיבי $interrups$, אלא רק מגיב להתרעות שיש להתחיל אותם. מצב זה שקול ללחכות שהשליח ידפוק בדלת במקום לצאת כל פעם החוצה ולראות אם הוא הגיע.
המצב רלוונטי גם לחומרה וגם לתוכנה, כלומר לסוגי $interrupts$ כגון;

- שגיאות תוכנה, כמו חלוקה ב־0...

- בקשה ל־$service$ של מערכת ההפעלה $(system \quad call)$
- בעיות מעבד אחרות כמו לופ אינסופי, מעבדים שמעדכנים אחד את השני או את מערכת ההפעלה...