מערכות הפעלה :

הרצאה 1 - הקדמה\מבנה מחשב על קצה המזלג :

ארכיטקטורת ואן נוימן נמצאת כמעט בכל מחשב. מכילה מודל בעל 3 חלקים :

- המערכת מורכבת מ4 תתי מערכות והן :
  - **זיכרון**
  - **ALU**– יחידת חישוב לוגית ואריתמטית.
  - **Control unit** – חלק מה -CPU ששולט על הכול.
  - **מערכות input/output**.
- התוכנית שלנו נטענת ורצה מתוך הזיכרון.
- יש לנו פקודות המהוות לנו את התוכנית והם רצות בצורה סדרתית פקודה אחר פקודה.

RAM– random access memory – ניתן להגיע לכל כתובת בזיכרון באותו זמן גישה O(1).

לכל תא בזיכרון יש כתובת. מערכת ההפעלה מבצעת על תא הזיכרון פעולות של **fetch** – לקחת משהו מהזיכרון או **store** - לאחסן משהו בזיכרון. לא ניתן לקרוא ביטים מהזיכרון אלא ניתן לקרוא מילה כיחידה אחת – 8 ביטים ברוב המקרים.

מילה יותר גדולה –> גישה ליותר מידע באותה יחידת זמן.

אחד הדברים שהכי חשובים בניהול הזיכרון זה ייצוג הכתובות. בשביל לייצג כתובת של תא כלשהו יכול להיות שנצטרך לשמור את הכתובת ביותר מתא זיכרון אחד – נצטרך הרבה מאוד ביטים על מנת לייצג את הכתובת.

מה שמעניין אותנו זה התוכן של תא הזיכרון.

מילה – 8 ביטים.

כתובת – N ביטים בדר"כ בכפולות של 2.

נשים לב להבדל בין מערכות של 32 ביט ו64 ביט. ב32 ביט בפועל נוכל לגשת רק ל4 גיגה זיכרון בכל פעם, בעוד שב64 ביט נוכל לגשת ל8 גיגה זיכרון בו זמנית.

גדלי זיכרון – חשוב למבחן.

זיכרון ה RAM הוא **נדיף** – ברגע שננתק את המחשב ממקור המתח כל המידע בו מתנדף. הוא מהיר ולכן יקר.

Fetch – לוקחים עותק של מקום בזיכרון ומעבירים אותו לרגיסטר מבלי לגעת במקור.

Store – לקחים מידע השמור ברגיסטר ולשים אותו בכתובת מסוימת. פעולה זו משנה את תוכן הזיכרון.

2 רגיסטרים שעוזרים לנו :

MAR – רגיסטר שמחזיק את הכתובת של המקום בזיכרון שנרצה לכתוב אליו.

MDR - מחזיק את התוכן אותו נרצה לכתוב\לקרוא מהזיכרון.

Signal – מפענח את סוג הפקודה.

מערכות I/O– לכל התקן כזה יש **controller** שתפקידו להפוך את האותות שהוא מקבל מהמחשב לאותו חשמליים שיפעילו את התקן הI\ O שלנו.

לcontroller יש מעיין buffer שהוא storage מקומי שהוא משתמש בו.

דוגמה לcontroller הוא הכרטיס מסך – הוא הcontroller של המסך, והוא יושב בmother board.

ALU – יש לו רגיסטרים משלו לצורך החישובים שהוא מבצע.

BUS– ערוץ תקשורת המשותף לכל הרכיבים וכל הנתונים עוברים דרכו.

התוכנית נשמרת בזיכרון. הControl unit בכל פעם טוען פקודה אחת מהזיכרון שלה ע"י הfetch באופן מעגלי ומבצע עליה decode ככה באופן מחזורי עד לפקודה האחרונה.

<u>לכל פקודה מספר שדות :</u>

8 ביטים המייצגים את הפקודה עצמה.

2 בתים = 16 ביטים המייצגים את כתובתו של X.

2 בתים = 16 ביטים המייצגים את כתובתו של Y.

מכאן השדות נשלחים לחישוב של הALU.

<u>2 גישות לארכיטקטורת מחשב :</u>

RISC– גישה המתבסת על סט מצומצם ופשוט של פקודות. יותר מהיר כי יש מעט פקודות שאנו מבצעים בצורה מאוד יעילה.

CISC– גישה המרחיבה את סט הפקודות, יותר קל למתכנת לתכנת.

<u>הגדרות נוספות :</u>

PC – תפקידו לשמור את הכתובת של הפקודה הבאה. מקדמים אותו כל פעם באחד אלא אם כן מתבצעת פקודת jump וכדו'.

Instruction register – רגיסטר שתפקידו לשמור את הפקודה הנוכחית אחרי שביצענו fetch ולפני שנבצע עליה decode.

-עד כאן חזרה על מבנה מחשב-

מערכת הפעלה – מתווכת בין המשתמש לחומרה.

מטרותיה :

- לנהל את גישת המשתמשים לחומרה.
- לנהל את משאבי המחשוב של מערכת המחשב – הקצאות, שחרורים וכו'
- ניהול הרצת תוכנות במקביל.

כשיש לנו מערכת מחשב שרצים עליה תהליכים של משתמשים שונים במקביל, יש שימוש יעיל יותר בחומרת המחשב.

דוגמה – כשאנו משתמשים בתוכנת הצייר, מתי אנו צריכים את מערכת ההפעלה?

- טעינת האפליקציה - טעינת התהליך שנוצר ע"י הרצת הקובץ paint.exe דורשת את מערכת ההפעלה – טעינת הקוד לזיכרון, יצירת התהליך וכו'. כמו כן בסיום העבודה עם האפליקציה יש לסיים את התהליך בצורה מסוימת.
- טיפול בהתקני הI\O תוך כדי הציור. מה שנקרא interrupts management.
- שמירה\פתיחה של קבצי צייר ישנים\חדשים.
- תקשורת בין תהליכים – העברת קובץ מהצייר לאפליקציה אחרת.
- הקצאה של זמן מעבד לתהליכים השונים.

יש לנו סוגים שונים של מערכות בהתאם לייעוד שלהן :

Supercomputer – המערכת הכי חזקה שיש היום בשוק. משמשת במערכות הביטחון ובאוניברסיטאות. היא רצה הכי מהר ממה שאנו מכירים היום. בדר"כ ישתמשו בה הרבה משתמשים בו זמנית. הרבה יותר מהיר מהmainframe.

Mainframe – מחשב חזק מאוד המשמש בעיקר לתהליכי ליבה ארגוניים או שרתים. נמצא בכל ארגון – יותר חזק מה PC הרגיל אך פחות חזק מהמודל הנ"ל. משמש בדר"כ לשרתי קבצים, DNS וכו'.

PC – מחשב אישי רגיל, מיועד לתמוך בהכול – בהרצה של חישובים מורכבים, ניהול והרצת מדיה וכו'. על כן הוא אינו מומחה בשום דבר והוא הרבה יותר איטי מהMF. יש לו בדר"כ משתמש אחד, לכן הפוקוס הוא על הנוחות של המשתמש בשימוש מול המחשב להבדיל מהמודלים הנ"ל שמתמקדים ביעילות ובנצילות.

Handheld – מכשירים ניידים כמו לפטופים וטלפונים. בדר"כ אינו מחובר לחשמל ואמצעי הI\O שלהם מאוד מוגבלים.

חזרה לשקף הקודם – המטרות :

- ניצול מקסימלי של משאבי החומרה ע"י הMF.
- הPC תומך בגמישות ובנוחות של המשתמש היחיד העובד עליו.
- במודל האחרון הדגש הוא על נתינת ממשק נוח להרצת אפליקציות למשתמש.

אנו בעצם נעים בין הציר של יעילות לציר של הנוחות. בדר"כ נמצא את עצמנו איפשהו באמצע כתלות בשימושים שנייעד למערכת המחשב שלנו.

נחלק את מערכת המחשב ל4 מרכיבים :

- חומרה
- מערכת ההפעלה
- תכניות של המשתמש – אפליקציות
- משתמשים – לא בהכרח בנאדם אלא אפילו מערכת מחשב אחרת שמשתמשת במערכת המחשב שלנו.

במצגת ישנה סכמה.

כבר אמרנו שאחד מתפקידי מערכת ההפעלה היא להקצות משאבים **resource allocator**.

Fair efficient – ניהול משאבים משותפים יעיל והוגן בין התהליכים. ("הוגן" בא לידי ביטוי בעיקר במערכות מרובות משתמשים– כל משתמש יקבל משאבים באופן הוגן, ולא תהיה עדיפות לתהליך מסוים)

כמו כן, מערכת ההפעלה היא **control program** – היא שולטת על ריצת כל התוכניות הקטנות ויודעת לבצע תיקונים במקרה הצורך.

מה כוללת מערכת ההפעלה?(מבחינת קבצים)

- אין הגדרה רשמית. יש הסוברים כי היא כוללת את כל מה שהספק שולח אלינו כשאנו מזמינים מערכת הפעלה.
- יש הסוברים שההגדרה המדויקת היא ה**kernel** – הגרעין, תהליך שרץ כל הזמן ברקע של מערכת המחשב. דוגמה לחלק מהגרעין הוא התהליך של ה**CPU scheduler** שתפקידו לתעדף את אחוז ה CPU שניתן לכל אחד מהתהליכים הרצים במחשב שלנו. כל שאר התהליכים שרצים נקראים **system programs** – כלומר יש תכניות נוספות שהגיעו יחד עם המערכת ההפעלה אך הם לא חלק מהkernel.

** ב1998 מייקרוסופט החליטה לצרוב את internet explore כחלק ממערכת ההפעלה ובכך מנעה תחרות בשוק. מכיוון שזה היווה ממש חלק מהמערכת הפעלה האפליקציה נחשבה להכי מהירה והכי מאובטחת. האינטרס של היצרן היא לדחוף למשתמש כמה שיותר תוכנות כחלק ממערכת ההפעלה.

מה קורה כשמדליקים את המחשב?

יש תכנית שנקראת **bootstrap program** המותקנת ב **ROM** של המחשב. לעיתים קוראים לה **firmware**. התוכנית הזו מועלית ברגע שהמחשב נדלק ומכינה את המחשב לקראת עלייה של מערכת ההפעלה. היא עושה אתחול לכל מרכיבי המערכת וממפה את כל רכיבי החומרה, דואגת שהזיכרון יהיה מוכן לפעולה ובגדול מביאה אותנו מצב שבו כל הkernel עלה.

בשלב הבא מערכת ההפעלה מחכה לevent שיזום המשתמש כמו תזוזות עכבר או הקמת תקשורת. מערכת ההפעלה שלנו לא יוזמת שום דבר היא מחכה שמשהו יקרה מצד המשתמש כל הזמן.

איך עובדת מערכת המחשב מ"מבט על" :

מורכבת מ CPU אחד לפחות או יותר ובנוסף **device controllers** (התקנים חיצוניים) המקושרים למעבד – יחידת העיבוד המרכזית, באמצעות ה**BUS** על מנת שהם יוכלו לגשת לזיכרון הראשי. כל אחד מהרכיבים הללו פועל במקביל ומתחרה על **memory cycles**.

**פעם היו טרמינלים טיפשיים שהיו מורכבים ממסך ומקלדת – זה היה בסך הכול ממשק שדרכו היה ניתן לעבוד מול המחשב המרכזי. היום בכל טרמינל ישנו מעבד.

כולם חשופים ויכולים לקרוא את מה שרץ בBUS.

Device controller – הוא כרטיס שאחראי על התקן כלשהו. כל התקשורת עם ההתקן מבוצעת באמצעות באפר – זיכרון לוקאלי שלו, כך שבסופו של דבר המידע מועבר לזיכרון המרכזי. הבאפר מספק לנו מהירות. מכיוון שמהירות ההתקנים אינם עומדת במהירות הגישה לזיכרון נקרא תחילה את כל הנתונים לבאפר ובבוא היום נעביר את הנתונים הללו לזיכרון המרכזי. כך אנו מאפשרים עבודה במקביל ל CPU וברגע שהנתונים יהיו קיימים בבאפר אז נוציא **interrupt** ל CPU ונעביר את המידע לזיכרון המרכזי.

ה CPU חייב שהנתונים יעברו קודם כל ל**main memory** על מנת שהוא יוכל לעבוד איתם.

<u>ארכיטקטורת מערכת המחשב :</u>

עד לפני 5-6 שנים לכל מחשב היה מעבד אחד **general purpose** (עם ליבה אחת), כיום יש לנו מעבדים עם כמה ליבות או מספר מעבדים מסוג general purpose. כמו כן יש מעבדים אחרים שמיועדים למשימות מסוימות כמו גרפיקה.

מערכות **multiprocessors** - בעלות 2 מעבדים או יותר המתקשרים ביניהם באמצעות אותו הBUS.

<u>יתרונות :</u>

Increased throughput - תוצרת גבוהה יותר.

Economy of scale – יותר זול לקנות 2 מעבדים פחות חזקים מאשר מעבד אחד חזק.

graceful degradation\Increased reliability – אם נהרס לנו מעבד בעל 4 ליבות נישאר ללא מעבד, מצד שני אם יש לנו 2 מעבדים כל אחד בעל 2 ליבות – נישאר עם מעבד אחד כך שהמערכת תעבוד טיפה יותר לאט עד שנתקן את המעבד השני.

יש 2 גישות לבניית מערכות מסוג זה :

- **נשתמש במעבדים בצורה אסימטרית** – רק מעבד אחד יוכל לגשת להתקני קלט-פלט והאחרים לא. זה נותן לנו שליטה ובקרה טובה יותר על התקני הקלט-פלט שלנו.
- **נעבוד בצורה סימטרית** – כל המעבדים שווים וכולם יכולים לעשות הכול. מערכת יותר מבוזרת.

Multicore – מספר מעבדים היושבים על אותה פיסת סיליקון וכך הם יכולים לתקשר בצורה טובה יותר. מבחינתנו, כשמדברים על עניני מערכת ההפעלה, מספר ליבות = מספר מעבדים.

האם תהליך יכול לרוץ על יותר ממעבד אחד בו זמנית? כן! כאשר יש לו כמה thread'ים.

הרצאה 2 :

תזכורת :

2 תפקידי מערכת ההפעלה :

- ניהול משאבים – **resource allocator**.
- הרצת התוכניות שלנו – **control program**.

עוד דרך להגדיל את עוצמת החישוב שלנו היא שימוש ב **cluster** - אשכול.

ניקח מספר מערכות מחשב שיעבדו ביחד והתקשורת ביניהם תהיה דרך הרשת. הcluster יעבוד בצורה מנוהלת וכל אחד מן המחשבים באשכול יוכל לטפל בjob.

הרעיון - נוכל להשתמש בארכיטקטורה זו כך שאחד מהמחשבים יהווה גיבוי חם לכל אחד מהמחשבים האחרים באשכול, כלומר בזמן אמת כשמחשב נופל, המחשב שהיווה הגיבוי ייקח את הפיקוד. חלק מהמחשבים יעשו כל הזמן **monitoring** על המחשבים האחרים, אותם מחשבים לא יעבדו ויכנסו לפעולה רק כאשר מחשב מתוך האשכול ייפול.

מימוש באופן סימטרי – כל המחשבים עובדים ומחלקים ביניהם את המשאבים.

מימוש באופן א-סימטרי – רק חלק מהמחשבים עובדים בפועל ושאר המחשבים יכנסו לפעולה ברגע שאחד מן המחשבים הפועלים ייפול.

3 המרכיבים העיקריים של מערכת המחשב שלנו :

- המעבד – יכולים לנו כמה כאלו\מספר ליבות.
- הdeviceים
- הזיכרון

כל הפקודות והמידע מאוחסנים בזיכרון שלנו. כאשר המערכת מריצה תכנית שלנו נבצע fetch לפקודה שיושבת בזיכרון ונביא אותה למעבד על מנת לבצע את החישוב.

החץ הדו כיווני של ה-**instruction exe cycle** בין המעבד לזיכרון– בשלב יצירת התהליך של התוכנית הפקודות נכתבות בזיכרון, מרגע זה רק נקרא אותם ולא נדרוס אותם. בזמן הטעינה נעלה את כל הקוד לזיכרון ובכל פעם נעביר פקודה למעבד. נקרא מידע מהזיכרון נבצע עליו חישוב כלשהו ונחזיר את הערך המעודכן לזיכרון.

לdevice נפנה אך ורק דרך ה-**device controller** לו יש רגיסטרים משלו ששומרים את המידע שלו. זהו החץ השמאלי היוצא מהמעבד ואומר לdevice controller מה הוא רוצה שיקרה.

החץ האמצעי של הdata בין הdevice לCPU - המידע עובר מהאחסון הלוקאלי של הdevice לאחסון המקומי שלנו במחשב. עלינו להביא את כל המידע שלנו לזיכרון על מנת שנוכל לעבד אותו. המעבד שולח הודעה לdevice על המידע שהוא מעוניין בו – המידע מועבר לזיכרון דרך חץ ה**DMA** ואז המעבד שואב אותו מהזיכרון ועובד עליו. ה- device מסתכל מה כתוב לו ברגיסטרים הלוקאליים שלו ויודע מה הוא צריך לעשות.

<u>דוגמה</u> – נרצה לקרוא\לכתוב ל HD במחשב שלנו. המעבד פונה לדיסק דרך הcontroller שלו ומבקש ממנו לקרוא משהו. הבקשה מתקבלת, מתבצעת פנייה לדיסק והמידע הרלוונטי נלקח ומועבר לרגיסטרים הלוקאליים. בסיום הטיפול ה HD שולח למעבד interrupt והטיפול מועבר אליו.

Interrupt – הודעה של הdevice למעבד - "קרה משהו" וזה מצריך טיפול מצדו.

הinterrupt מעביר את השליטה מהמעבד לרוטינה שצריכה לקרות – **interrupt service routine**.
דוגמאות – תזוזת עכבר, הקשה על כפתור במקלדת וכו'. במצב זה המעבד יעזוב הכול ויטפל ב-interrupt – אם קרה משהו כנראה שזה דורש טיפול ותגובה מידית.

מכיוון שעל המעבד לעזוב הכול עליו לשמור את הכתובת של הפקודה שבה הוא עצר וכמו כן את תוכן הרגיסטרים עד כה על מנת שבסיום הטיפול הוא יוכל לחזור לנקודה שבו הוא עצר. הדבר הזה הוא **overhead** המעבד שומר לרגע את כל הנתונים בצד וזה דורש ממנו זמן רב לעיתים. לרוב בעת הטיפול בinterrupt המערכת תבצע disable לinterruptים נוספים שמגיעים בינתיים. בסיום הטיפול בinterrupt זה, הdeviceים ישלחו שוב interrupt למעבד. המעבד לא מקבל בזמן זה בקשות נוספות. הdevice יודע כאשר המעבד חסם את הinterrupt שלו והוא ימשיך לנסות כל כמה ננו-שניות.

Trap – מאוד דומה לinterrupt אך אינו נוצר ע"י אות חשמלי שנשלח מהcontroller אלא ע"י תוכנה שרצה על המחשב כמו למשל כשמתקבלת שגיאה.

<u>שיטות לטיפול בinterrupt :</u>

תחילה יש להבין מה מהותו – מי יצר אותה ולמה. יש 2 שיטות לעשות זאת :

- **Polling** – נתחקר את הסביבה – מי ולמה שלח. ברגע שהמעבד מקבל interrupt המעבד פונה לזיכרון ומעביר את השליטה לקוד <u>גנרי</u> שיושב בזיכרון והוא מתחקר מה קרה ויודע לטפל בכל סוג של interrupt.
- **Vectored interrupt system** – נחזיק ווקטור במערכת ההפעלה. בשיטה זו נחזיק רוטינות בזיכרון שתפקידן לטפל בInterruptים שונים. הווקטור יחזיק במקום ה-i את הכתובת בזיכרון בה שמורה הרוטינה שאמורה לטפל בinterrupt ה-i. כל איבר בווקטור הוא פוינטר לרוטינה בזיכרון שאמורה לטפל בסוג הזה של הinterrupt.

<u>יתרונות השיטה הראשונה</u> – רוטינה אחת גנרית שיודעת לעשות הכול ותופסת פחות זיכרון.
עליה לתחקר את הסביבה על מנת לדעת באילו נסיבות להריץ את הקוד הגנרי.

<u>יתרונות השיטה השנייה</u> – הרבה יותר מהירה כי אין את הפן של התחקור כדי להבין מה קרה – פשוט נקראה לרוטינה המתאימה שתטפל במה שצריך. נדע לרוץ מיד למקום המתאים ולהריץ את הקוד הרלוונטי לטיפול.

<u>המעבד יכול להיות באחד משני מצבים :</u>

- טיפול בתהליך של המשתמש.
- טיפול בInterrupt שרק הגיע.

<u>ה device יכול להיות באחד משני מצבים :</u>

- לא עושה כלום – idle, פנוי.
- מעביר אינפורמציה.

בdevice controller של ה i/o device נייצר את הבקשה המתאימה שהגיעה מהמשתמש. הוא מסיים ושולח interrupt. המעבד עוצר בנקודה זו ומגבה את כל המידע שלו ומריץ את קוד הטיפול ב-interrupt.

שיטות לבקשת I\O :

- **השיטה הסינכרונית** - נבקש מהdevice בקשה ולא נתקדם בהרצת תכנית המשתמש כל זמן שהdevice לא שלח interrupt שהמידע שחיכינו לו מוכן. התוכנית מחכה למידע מסוים על מנת שהיא תמשיך לרוץ.
- **השיטה הא-סינכרונית** – שיטת "שגר ושכח" - מתאים למקרים שבהם נבצע חישוב מאוד ארוך וכל כמה שניות נשלח הודעה למסך של מה שחישבנו ולא אכפת לנו אם ה device controller סיים את הטיפול בקשה או לא נוכל להמשיך להריץ את התוכנית של המשתמש.
בזמן שבו הבקשה מועבדת נוכל להמשיך להריץ את התוכנית של המשתמש.
איזה מרכיב נוסף עלינו לתחזק על מנת לעבוד בשיטה זו? – עלינו לתחזק תור בקשות שכן עלול לקרות מצב שהגיעה בקשה נוספת לפני שסיימנו לטפל בבקשה הנוכחית. עלינו לתחזק תורים לכל device – לשרשר את הבקשות שמגיעות אליו – **device status table**. נשמור מה הבקשה שהגיעה, מה מהותה וממי היא הגיעה.

על הפקודות להיות בזיכרון על מנת שהמעבד יוכל לבצע אותן. על המידע להיות גם הוא בזיכרון על מנת שנוכל לעבד אותו – לאחר העיבוד הוא חוזר שוב לשבת בזיכרון.
ניהול הזיכרון – עוסק בהחלטה מה ישהה בזיכרון בכל רגע ורגע. זהו מרכיב קריטי לניצולת של המעבד ובעל השפעה מכרעת על זמן התגובה למשתמש.

תפקידי המערכת – שקופית 31.

Main memory – הזיכרון הראשי אליו המעבד יכול לגשת ישירות. הזיכרון הראשי נדיף ואילו המשני קבוע והמידע בו ישמר לאורך זמן גם לאחר שנכבה את המחשב.

אחסון מידע בדיסקים – מידע שנצטרך לשמור לתקופת זמן ארוכה.

התקנים נוספים שמהירותם פחות חשובה ואמינותם חשובה יותר ישמרו ב – **WORM** מדיה שנכתוב אליה בפעם אחת אך נוכל לקרוא ממנה הרבה פעמים. נשתמש במערכות שיהיה לנו חשוב לשמור היסטוריה של מה שהיה קודם. דוגמה לכך הוא ה-ROM-CD.

היררכיית הזיכרון – הרגיסטרים נמצאים בראש הפירמידה. ככל שעולים בפירמידה העלות גבוהה יותר פר יחידת מידע, המהירות גבוהה יותר והזיכרון נדיף.

אם יהיה לנו התקן יותר מהיר ויותר זול נחליף אותו בזה שמעליו בפירמידה.

מערכת ההפעלה עושה דבר נוסף לניהול הזיכרון והוא ניהול מערכת הקבצים. מערכת ההפעלה מייצרת מעיין "ישות" שנקראת קובץ וישות נוספת מעל הקובץ שנקראת מחיצה. על המערכת ההפעלה לתמוך במערכות קבצים שונות על מנת שכשנחבר כונן חיצוני הוא תוכל לטפל בו.

מערכת ההפעלה יודעת לייצר ולמחוק קבצים ומחיצות.

מערכת ההפעלה יודעת לתת לנו כל מיני **primitives** לביצוע מניפולציות על הקבצים כמו מחיקה, יצירת קובץ חדש וכו'

מערכת ההפעלה יודעת לעשות מיפוי של הקבצים לstorage עצמו.

Cache – שמירת מידע שנשתמש בו בתדירות גבוהה ע"י כך שניקח אותן מרמה תחתונה יותר בפירמידת הזיכרון ונעביר אותו לרמה גבוהה יותר. כאשר נרצה לקרוא נתון תחילה נבדוק אם הוא קיים בcache ורק לאחר מכן נרד לרמות התחתונות בהיררכיה.

המידע מועתק לזיכרון היותר מהיר. תמיד הcache שלנו יהיה יותר קטן מהזיכרון עליו נעשה את ה-cache, אחרת היינו שומרים את כל הזיכרון בו, למעט מהמקרה שהזיכרון volatile.

במצגת ישנה טבלה ובה הגדלים של אמצעי הstorage השונים, זמני הגישה שלהם, כמה מידע נוכל להעביר, מי עושה cache למי ומי מנהל אותו.

אחת הבעיות הגדולות שלנו עם הcache היא שמרוב שאנו מייצרים לעצמו עותקים של אותו מידע בהתקני אחסון שונים עלולה להיגרם לנו בעיית סינכרוניזציה. עלול להיגרם מצב בו מעבד כלשהו מנסה לגשת לנתון מסוים ב-main memory מבלי לדעת שאותו הנתון מתעדכן כרגע ברמה אחרת של הזיכרון. בעיה זו תהיה מאוד חמורה במערכות של clusterים. לכן מנסים לגרום לחומרה לדעת מה שמור בcacheים השונים במחשב – החומרה מנסה לטפל בזה.

DMA– מקשר בין ה device ל memory – מעביר מידע ישירות מה device controller לזיכרון ללא התערבות של המעבד. על איזו התקנים נרצה שזה יקרה? ב-HD! כי ה HD קורא המון נתונים ולא נרצה להודיע כל פעם למעבד ע"י interruptים על כל מידע שנרצה לגשת אליו. ה DMA יאגור בבאפר הלוקלי שלו הרבה מידע ויעביר אותו ישירות לזיכרון ואחת לכמה בלוקים כאלה הוא ישלח interrupt למעבד ויודיע לו שיש לו מידע מוכן, זה במקום לייצר interrupt על כל פיסת מידע שנקרא מהדיסק ובכך נפריע לפעולה הסדירה של המעבד.

מתי זה לא יהיה שימושי? במקלדת! כי נרצה לקבל מיד את תגובת המעבד כאשר נלחץ על מקש כלשהו במקלדת. במקרה הזה אנחנו כן רוצים לייצר interrupt עבור כל פיסת מידע שנקרא.

<u>עוד כמה מילים על מבנה מערכת ההפעלה :</u>

אם נרצה להביא לניצולת מקסימלית של המעבד שלנו, חשוב שבכל רגע נתון יהיה תהליך שיכול לרוץ. תהליך שמחכה לקלט\פלט הוא תהליך שאין לו מה לעשות.

נשתמש בקונספט שנקרא **multi-programing** – כמות התהליכים שרצים במערכת גדולה או שווה לכמות המעבדים או הליבות שיש לי במערכת. כלומר יהיו תהליכים שיחכו ובכל רגע נתון נחליט מי יהיה התהליך שישב על המעבד וירוץ. בפועל התהליכים שבאמת מריצים פקודות יהיו כמספר המעבדים או הליבות שישי לנו במערכת.

כל job בציור הוא תהליך שמחכה או רץ. הם כולם יושבים תחת מערכת ההפעלה.

Timesharing\multi-tasking – נשתמש בקונספט הנ"ל ונשמור על זמן תגובה נמוך למשתמש כך שזמן התגובה שהמשתמש ירגיש יהיה פחות משנייה אחת. המשתמש ירגיש כאילו כל מערכת המחשב עובדת לטובתו ושהתהליך שלו כל הזמן רץ ללא delay. נשמור על כך שמעבד יגיע לכל אחד מהתהליכים ללא יותר משניה אחת בנקודת הזמן שבו הוא קיבל את הפניה מהמשתמש.

CPU scheduling - אותו המרכיב שקובע מי ירוץ בכל נקודת זמן על המעבד, הופך להיות מאוד מאוד מרכזי.

Swapping – העברת תהליכים מהדיסק לזיכרון ולהיפך.

<u>בעיה</u> – אין לנו די מקום בזיכרון לאחסון כל התוכניות שלנו. מה שפותר בעיה זו הוא ה **virtual memory** עליו נלמד בהמשך.

מה עוד עושה מערכת ההפעלה – נסתכל על תפקיד הcontrol program - שולחת לנו הודעות שגיאה במקרה של תקלות. במקרה של לולאה אינסופית או dead lock מערכת ההפעלה לא תדע איך להתמודד.

נרצה להימנע ממצב שתוכניות נוגעות בקטעי קוד של תכניות אחרות. נשתמש בפקודות שהן **privileged** – פקודות שרק מערכת ההפעלה יכולה להריץ/פקודות שיכולות לרוץ רק בkernel mode.

מערכת ההפעלה שלנו יכולה לרוץ ב-2 מצבים :

- **User mode**
- **Kernel mode**

על מנת לתחזק אותם נשמור ביט בחומרה שישמור לנו על איזה מצב אנו רצים כרגע.

System calls – פקודות שיכולות לרוץ רק במצב של kernel mode.

דוגמה מהמצגת – תהליך של משתמש מבצע חישובים עד להגעה לפקודה שהיא privilege. במצב זה נשלח trap שישנה את הביט הנ"ל על מנת שנעבור לkernel mode ונוכל להריץ את הפקודה ע"י קריאה לsystem call.

שאלה – אילו מהפקודות במצגת היינו רוצים שירוצו בkernel mode?

- קריאת שעון – אין צורך.
- לנקות חלק מהזיכרון – זיכרון שלא משתמשים בו יותר.
- יצירת trap – אין צורך כי כל הרעיון שכל תכנית תוכל ליצור אותם.
- סגירת interrupt.
- עדכון של entries בdevice status table .
- לעבור מuser mode ל kernel mode – נרצה להיות מסוגלים לבצע systems calls או לשלוח trapים.
- גישה לdevice i\o.

program – אוסף של פקודות – ישות פאסיבית. יש לה פוטנציאל להיות מורצת. ברגע שאנו מריצים תכנית נוצר process. על מנת שתהליך יוכל לרוץ עליו לקבל משאבים כמו זיכרון, זמן מעבד וכו'. כמו כן צריך לאתחל אותו עם איזשהו data. כשהתהליך מגיע לנקודת הסיום שלו הוא מודיע זאת למערכת ההפעלה ע"י exit וזה הסימן למערכת ההפעלה לבצע ניקיון – **reusable** – לקחת חזרה את כל המשאבים שהוקצו עבור תהליך זה.

משאבים שאינם reusable כלומר אינם ניתנים לשחזור הם זמן מעבד למשל.

לכל תהליך יש PC אחד ויחיד משלו, אלא אם הוא בנוי ממספר Threadים - לכל אחד מהthreadים השונים יהיה program counter משלו.

<u>אילו שירותים נותנת מערכת ההפעלה בהקשר של ניהול תהליכים?</u> – יצירת תהליכים, עצירת תהליך, הריגה שלו והשהייה שלו.

שקופית אחרונה – שאלות שיכולות להיות במבחן על המבוא.

-סוף הפרק הראשון-

הרצאה 3 :

נדבר על השירותים שמספקת מערכת ההפעלה למשתמשים שלה, לתהליכים שרצים עליה ומערכות אחרות. כמו כן נדבר על ארכיטקטורות שונות לעיצוב המחשב.

נסתכל על מערכת ההפעלה ממספר נקודות מבט :

- השירותים שהיא מספקת.
- הממשקים שהיא מספקת.
- מרכיבי המערכת והקשרים ביניהם.

השירותים שהיא מספקת – התרשים במצגת מתאר מערכת מחשב. מערכת ההפעלה ממוקמת בתרשים בחלק הכחול כולל החלקים בתוכה באפור. הsystem calls זו הצורה שאנו פונים למערכת ההפעלה על מנת שתבצע עבורנו דברים שלנו אסור לעשות. על מנת לגשת לכל הפונקציונאליות הזאת נצטרך שמערכת ההפעלה תעשה זאת עבורנו ע"י פקודות privilege. אנו מעבירים את ה mode bit מ user mode לsystem mode.

פונקציונאליות שבאה לסייע למשתמש :

User interface יכול להיות בהרבה מאוד צורות ויכולות להיות כמה סוגים ממנו על אותה מערכת ההפעלה כמו :

- **CLI** – command line – מאפשר לנו הזנת פקודות בצורה טקסטואלית ותרגום שלהם לsystem calls. כמו כן יכולים להיות לנו מספר מתרגמים **interpreters** . שם נוסף לכך הוא ה**shell** ונוכל לעבוד עם כמה shellים שונים. הפונקציונאליות הבסיסית שלו היא לקחת פקודה להריץ אותה.
- **System program** – תכניות שהגיעו עם מערכת ההפעלה כמו סייר הקבצים, משחקים מובנים וכו'.

2 שיטות למימוש CLI:

- הפקודות מובנות בתוך הממשק. המתכנת בונה תכניות שיודעת להיתרגם לפקודות מובנות וקיימות.
- הפקודה היא שם של תכנית היושבת על הדיסק והפקודה שנזין תריץ לנו תכנית באותו השם. ב-unix כל הפקודה היא קובץ, למשל הפקודה rm היא קובץ תכנית – מערכת ההפעלה תחפש את קובץ המימוש של הפקודה הזו על מנת לבצע אותה.

מה יותר טוב? עבור תחזוקה פשוטה יותר נעבוד באופציה השנייה כי כל מה שנצטרך לעשות זה לשנות את הקובץ של הפקודה הספציפית במקרה ונרצה לשנות פקודה מסוימת ללא תלות בשאר הפקודות. בשיטה הראשונה הכול גנרי ומשפיע אחד על השני.

היתרון באופציה הראשונה – מהירה יותר כי אין צורך לטעון לזיכרון את התכנית של פקודה. כמו כן יש לנו הרבה פחות קוד.

- **GUI**– ממשק גרפי שמוצג למשתמש עם תפריטים, כפתורים וכו'

כיום הרבה מערכות כוללות גם CLI וגם GUI. מי שמתמצא בסינטקס של הפקודות בCLI ישתמש בדרך כלל ב CLI כי זה יותר מהיר. למשל העברה למספר קבצים – מי שמתמצא יוכל לעשות זאת ע"י ה CLI בפקודה אחת ואילו ב GUI קובץ-קובץ ע"י לחיצת כפתור בכל פעם. המשתמשים היותר מתוחכמים יעדיפו את הCLI.

פונקציונאליות שבאה לסייע למשתמש – דוגמאות נוספות:

- הרצת תכניות.

תחילה נטען את התוכנית לזיכרון, נריץ אותה ובסיום ההרצה או שריצתה תסתיים מעצמה בצורה טבעית או שתסתיים בצורה לא טבעית בגלל שגיאה, במקרה זה המערכת ההפעלה תצביע על השגיאה ותטפל בה.

- טיפול בO\I.
- טיפול בקבצים.
- תקשורת בין תהליכים.
- טיפול בשגיאות. לדוגמה טיפול בחומרה – מדפסת נתקעה וכו'.

Correct and consistent computing – מערכת ההפעלה תיתן פתרון נכון ועקבי.

Debugging facilities – מערכת ההפעלה נותנת לנו כלים על מנת להבין בדיוק מה קרה בתהליך שהייתה בו שגיאה.

על מנת להבטיח שפעילות המערכת תהיה יעילה היא :

- **מקצה לנו משאבים** – לחלק מהמשאבים נשתמש בקוד הקצאה מאוד מאוד פשוט כמו הראשון שמבקש מקבל – דרך כלל עבור התקנים שמחוברים למחשב כאשר מספר תהליכים מבקשים לגשת להתקן מסוים כמו מדפסת. למשאבים מסוימים ממש נכתוב קוד מסובך שינהל את ההקצאה שלהם כמו זיכרון.
- **Accounting** – קבצי לוג אליהם מערכת ההפעלה כותבת כל הזמן מי ביקש לגשת לאיזה משאב.
  למה זה חשוב?
  - **tuning** - היכולת לקנפג ולנהל את מערכת ההפעלה בצורה טובה יותר.
  - **billing** - מערכת המחשב מסתכלת מי השתמש באיזה משאבים במשך כמה זמן ולפי זה יודעת לקבוע את העלות עבור כל תהליך.

הבטחת הפעילות היעילה של המערכת תתבצע על ידי **Protection and security** – גישה תקינה למשאבי המערכת והגנה עליהם מגישה זדונית.

הsystem calls מהוות את ממשק התכנות של המתכנת. ישנן דוגמאות לפקודות כאלה בUNIX.

לכל פקודה כזו יהיה מספר ואז שנקרא לה יהיה לנו טבלה והמערכת הפעלה תיגש למספר המתאים בטבלה ולפיו היא תדע איזה פקודה עליה להריץ.

למשל – בביצוע הפקודה open – תחילה נעבור לkernel mode ניגש לטבלה ונשלוף ממנה את המספר המתאים לפקודה ונריץ אותה.

עד כמה נפוץ השימוש בsystem calls? – נניח כי המשתמש רוצה להעתיק את תוכנו של קובץ מקור לקובץ יעד כלשהו. בכמה system calls נשתמש (בסך הכול לאו דווקא שונים)?

בשלב ראשון על המשתמש לומר לנו מה קובץ המקור ומה קובץ היעד. המשתמש יכתוב לנו זאת ואנו נפנה לdevice על מנת לשלוף את המידע – דורש הרבה מאוד system calls לטיפול בדברים האלה.

כעת עלינו לעשות open שזה system call לשני הקבצים. כמובן שיכולות להיות שגיאות – במקרה זה נצטרך לחזור למשתמש ולשאול אותו מה ברצונו לעשות – system calls נוספים.

כעת נרצה לקרוא ולכתוב בעזרת לולאה. גם כאן ייתכנו שגיאות כמו מקום שנגמר, הגעה לסוף הקובץ מוקדם מידי וכו' – הטיפול בהן נעשה באמצעות system calls.

סגירת הקבצים – דורש system calls .

גם פקודת סיום exit של התכנית היא בעצמה system call.

יש 319 סוגים שונים של system calls בלינוקס ובמערכות שונות 330 סוגים.

API– הממשק שמערכת ההפעלה נותנת לנו על מנת לקרוא לsystem calls השונים. זה אוסף של פקודות. דוגמה – נבצע הדפסה למסך באמצעות הפקודה printf - היא לא באמת נקראת כך זהו ה API שלה, השם האמתי שלה הוא write וזו בעצם עטיפה של הפקודה לצורך נוחות של המשתמש.

<u>העברת פרמטרים לSystem calls :</u>

הרבה פעמים נצטרך להעביר פרמטרים לsystem call. איך נעשה זאת? – באמצעות רגיסטרים אליהם נכניס את הInput שלנו. הבעיה היא שאנו מוגבלים בכמות המידע שאנו יכולים לשמור ברגיסטרים, לכן נוכל להעביר <u>כתובת של מקום בזיכרון</u> (בלוק) שממנו והלאה המידע שנרצה להעביר שמור. אפשרות נוספת להשתמש במחסנית, בעזרת פעולות סטנדרטיות (pop, push). יש דוגמה בשקופית 26.

<u>דוגמאות למערכות הפעלה:</u>

MS-DOS - דוגמה למערכת הפעלה (ישנה) שעובדת עם פקודה אחת בכל רגע נתון – **single-tasking**. היא הופעלה בעיקר במחשבים דלי משאבים. איך היא עבדה? יש שתי תמונות זיכרון בדוגמה בשקופית 32 – a ו- b.

<u>בתמונה a:</u> יש את **הkernel** שרץ תמיד, מעליו יש את **command interpreter** (מפרש הפקודות), ואז כשהיו מריצים פקודה- והדיסק היה מריץ תכנית, ברגע שהcommand interpreter היה מפרש את הפקודה, הוא היה מכווץ את עצמו (הוריד מהזיכרון חלקים שלא קשורים להרצת הפקודה) על מנת לפנות כמה שיותר מקום בזיכרון לתהליך שרץ (<u>תמונה b</u>). ברגע שהתהליך סיים- הוא טוען את עצמו מחדש לגודלו המקורי על מנת לתת למשתמש את הפונקציונליות המלאה של הCLI.

FreeBSD – סוג של Unix. דוגמה למערכת הפעלה שיודעת לעשות **multitasking** – כמה תהליכים שרצים ושייכים פוטנציאלית לכמה משתמשים. כאן בתמונה לצד הkernel שתמיד נמצא, רואים כמה תהליכים ואת הinterpreter. לכל משתמש נפתח CLI לפי בחירתו. התהליכים פה נוצרים בעזרת פקודת fork().

System programs – תכנית שהגיעה יחד עם מערכת ההפעלה, אך הן אינן חלק מהkernel. למשל "צייר". אילו תכניות נמצא בדרך כלל במערכת ההפעלה?

- **תכניות לטיפול בקבצים** כמו סייר הקבצים לביצוע – מחיקה, העתקה וכל המניפולציות על קבצים ומחיקות.
- **תכניות שנותנות מידע וסטטוס** – שעון, תאריכון, תכונית שתראה לנו כמה מקום יש בדיסק, מי הusers המחוברים למערכת.
- **תכניות שמטפלות בתוכן קבצים** – כמו notepad.
- **תכניות registry** – הסבר למטה.
- **תכניות שתומכות בכתיבת תוכנה**- קומפיילרים, interpreterים, לינקרים וכו'.
- **תכניות לתמיכה בתקשורת** - בין מחשבים.

Registry – תכנית שמחזיקה את כל מידע הקונפיגורציה על מערכת ההפעלה שלנו (איך אמורה לרוץ, איך הscheduler שלה אמור לעבוד וכו').

באיזה שפה כתובה מערכת ההפעלה שלנו? – יש מגוון אפשרויות.

היתרון בשימוש בשפות עיליות – העברתן ממערכות מחשב אחת לשנייה, אך משלמים במהירות.

אמולטור/אמולציה – דימוי של מערכת הפעלה כמו מכונה וירטואלית. פעם היו כל מיני **אמולטורים** שהיה אפשר להריץ עליהם דברים.

מבנה מערכת ההפעלה:

מה הארכיטקטורה עצמה של מערכת ההפעלה שלנו? – נבחין בין המטרות של המשתמש למטרות מנהל המערכת:

- **המשתמש** ירצה מערכת הפעלה מהירה, נוחה לשימוש, קלה ללימוד, בטוחה – קשה יהיה לחדור אותה.
- **כמנהלי המערכת** חשוב לנו שdesign יהיה נח לשינויים, שיהיה קל להעביר את השינויים למשתמש מאחורי הקלעים, קלה לתחזוקה, גמישה, חסינת באגים, אמינה, יעילה.

דוגמאות לעיצובים של מערכות הפעלה:

1. במצגת יש תרשים ובו המבנה של מערכת ההפעלה **MS-DOS**. זהו מבנה על בסיס שכבות. מה שמפחיד אותנו הן הגישות הישירות משכבת החומרה לשכבת האפליקציה – גישה ישירה שכזו יכולה לגרום לנו לבעיות ולפגוע במערכת ההפעלה. לדוגמה אפליקציות יכולות לגשת ישירות לדיסק (תוך עקיפה של מערכת ההפעלה) ולכתוב אליו או יותר גרוע- למחוק אותו וזה יכול לגרום לבאגים.

2. בתרשים של **UNIX** (שעובדת multitasking) יש לנו שכבה אחת שהיא הkernel – כאשר מערכת ההפעלה מחולקת ל2 חלקים - הגרעין ובו כל הפונקציונאליות (הוא זה שקורא לחומרה), וה- system programs שיכולנו להריץ אותם. הבעיה – הרוב מרוכז בkernel מה שאומר שיהיה למשתמש מאוד קשה לבצע שינויים.

3. דוגמה נוספת היא – **ארכיטקטורה של שכבות** – כל שכבה יכולה לפנות לפונקציות שנמצאות בשכבה שמתחתיה בלבד. השכבה הנמוכה ביותר 0 תהיה שכבת חומרה. יתרון -

למשתמש יש גישה לרמה הראשונה והוא לא יכול להגיע עמוק מבחינת הפונקציונאליות.
היתרונות - המערכת הפעלה יכולה לשמור על עצמה בצורה יותר טובה. נשים את הדגש מבחינת האבטחה על מה שקורה במערכת ברמות התחתונות. יתרון נוסף הוא הdebugging - ההפרדה המלאה – אם יש לנו באג אזי הוא נמצא במקרה הכי גרוע באותה הרמה ומטה. לכן קל לנו לעשות debug בצורה מאוד קלה.

החסרונות – קשה לנו לבצע את ההפרדה מבחינת השכבות – החלוקה מאוד מאתגרת מבחינת חלוקה נקייה של הפונקציונאליות לשכבות. כמו כן ישנה בעיית יעילות – כל שכבה קוראת לשכבה מתחתיה עד שנגיע לרמת החומרה, כלומר מתבצעות הרבה מאוד קריאות פנימיות.

4. דוגמה נוספת היא **Microkernel -** – גרעין "רזה" - נרצה לבנות kernel כמה שיותר קטן שיש בו רק את הדברים ההכרחיים. ניקח את כל החלקים של הפונקציונאליות שלא קריטיים למערכת ההפעלה ונעביר אותם לuser mode. כל התקשורת בין החלקים ב user mode תעבור דרך messaging שיעברו דרך הkernel mode שם יהיו תקשורת בין התהליכים, ניהול זיכרון, וניהול זמן המעבד.

יתרונות – קל להרחבה (כי יש פחות קוד שצריך לשנות), קל לאבטחה – ככל שנרחיק יותר קוד מkernel mode ככה נרחיק אותו מהחומרה, כלומר הוצאה של חלק מהdevice וה-program החוצה – תהיה מאובטחת יותר.

חסרונות - החיסרון הוא בביצועים. הביצועים יורדים, כיוון שפעולות מסוימות בין שני devices שונים צריכים לעבור דרך הkernel.

5. דוגמה נוספת היא **Modules** – ארכיטקטורת מודולים. במקום לארגן את הכול בשכבות, ניקח כל תת מערכת והוא יהיה מודול שעומד בפני עצמו – ננהל אתו במודול עצמאי. נוכל להעלות מודולים כרצוננו. המודולים יהיו חלק מהליבה של הkernel. בתרשים בשקופית 47, כל עיגול זה מודל שמחובר לkernel.

וירטואליזציה : (Virtual Machine)

וירטואליזציה היא תוכנה שנותנת לנו interface שמייצר אצלנו את ה"הרגשה" שאנחנו עובדים מעל חומרה\מערכת הפעלה אחרת מזו שבאמת מותקנת במחשב.

ניקח תוכנה ונתקין אותה על המחשב שלנו – או ישירות על החומרה או על מערכת ההפעלה שלנו והתוכנה הזו תיתן לנו ממשק הזהה למערכת הפעלה כבחירתנו.

איך נתקין את הVM-ware?

שיטה אחת - נוכל לקחת את החומרה שלנו ולהתקין מעיין שכבה ישירות על החומרה והיא תאפשר לנו להריץ מכונות וירטואליות שונות שכל אחת מהן תריץ interface של מערכת הפעלה אחרת. יש לנו הפרדה מלאה בין המכונות. כל המשאבים מופרדים כאשר מבחינת התהליכים הם רואים מערכת הפעלה מופרדת לגמרי.

שיטה שנייה – ניקח מחשב שיש עליו כבר מערכת הפעלה. על מערכת ההפעלה נריץ תכניות שונות כמו מכונה וירטואלית על מערכת הפעלה שונה לגמרי עליה נריץ תכניות המתאימות למערכת הפעלה זו.

למה זה טוב?

- אנחנו מצליחים להשיג הפרדה מלאה ברמת התוכניות שרצות. תכנית אחת לא מפריעה לתכנית אחרת. מבחינת כל תכנית היא עובדת על מחשב נפרד לחלוטין.
- חיסכון בחומרה, פחות מחשבים – ניתן להריץ מספר מערכות הפעלה על אותו המחשב במקום להשתמש במספר מחשבים. נתקין מכונה וירטואלית לכל מערכת הפעלה שנרצה.
- רווח לתכניתן – נכתוב קוד שונה לכל מערכת הפעלה אך לא נצטרך לרכוש כלים שונים שבודקים את הקוד.
- אם התוכנית שלי קורסת לעיתים קרובות – היא תגרום למכונה לקרוס אבל לא תשפיע על שאר התהליכים הרצים במחשב. במקרה הכי גרוע נעלה פשוט את המכונה הווירטואלית מחדש.

JVM – עובדת בצורה מאוד דומה. יש לנו את ה java interpreter שיודע לקחת פקודות של JAVA ולתרגם אותן לsystem calls. הפקודות של ה JAVA מתורגמות לsystem calls המתאימות למערכת הנושאת. אם נריץ JVM מעל הווינדוס הפקודות יתורגמו לsystem calls של הווינדוס.

הרצאה 4 :

תהליכים (Processes) – מתוך מצגת 3

המערכות של היום מאפשרות הרצה והעלאה של מספר תהליכים במקביל. כתוצאה מכך, כמות המשאבים לכל תהליך קטנה, ונדרשת שליטה הדוקה יותר בכל אחד.

מטרת השיעור היא להציג את ה"תהליך" ולתאר מאפיינים שלו.

בקורס נשתמש במונחים **job** - מתייחס לbatch system ו – **process** - בד"כ למערכות שהן time shared , באופן חלופי (יש להם אותה משמעות).

סוגי תהליכים:

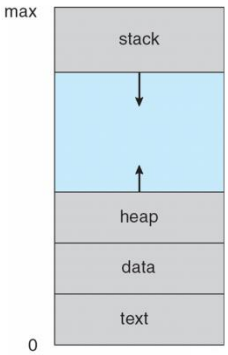
1. תהליך של מערכת ההפעלה
2. תהליך של המשתמש

תהליך הוא "תכנית בהרצה". הרצת התהליך היא סדרתית (הרצת פקודה אחר פקודה). התהליך כולל: PC, stack, heap, text ו-data section.

מצורפת תמונת הזיכרון של התהליך. הכתובות שכתובות הן כתובות וירטואליות.

מרכיבי התהליך:

1. **Text** – נשמרים בו כל הפקודות של התכנית , שממנה נוצר התהליך, וכל ה- raw data.
2. **Data** – נשמרים בו כל המשתנים הגלובליים.
3. **Heap** – נשמרים בו כל ההקצאות הדינמיות (כל פעם שנעשה new). גודלו לא קבוע.
4. **Stack** – נשמרים בו כל המשתנים הלוקאליים.



בדוגמה הקטנה למטה – המשתנה x נוצר בstack כי הוא משתנה לוקאלי. ההקצאה נוצרת בheap.

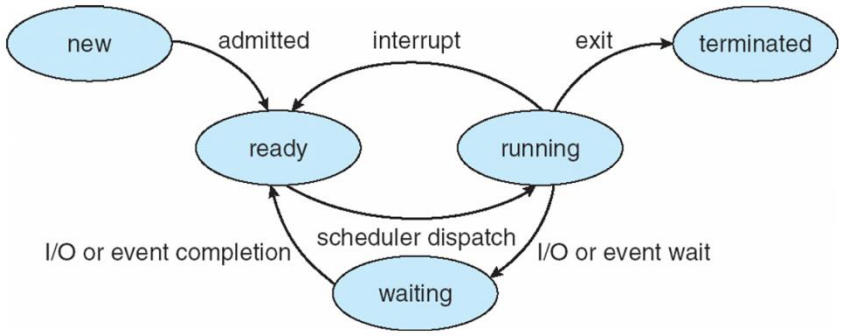
מאותה תכנית ניתן לייצר כמה תהליכים, לכל אחד תמונת זיכרון משלו. לדוגמה בשקופית 8, ניצור Program ולו שני תהליכים. האם בשניהם יהיו heap, data, text, stack?- כן. אז מה ההבדל?
הtextים יהיו זהים בגודלם ובתוכנם. לעומת זאת, הPC יהיה שונה ביניהם. הdata זהים בגודלם, ולא בהכרח זהים בתוכנם (בנקודה ששניהם עלו הם בוודאות יהיו זהים בתוכנם, אלא אם רצו עם inputים שונים). לגבי ה heap וה -stack הם יהיו שונים בגודלם ובתוכנם (למעט בהתחלה, כי הם דינאמיים לפי המיקום של התהליך בקוד).

Process State - לכל תהליך יש state שהוא נמצא בו – המצב שמתאר אותו. יש חמישה מצבים (יש מערכות הפעלה שיש יותר או פחות, אבל זו החמישייה המאפיינת):

1. **New**- התהליך נוצר. מצב "מומנטרי" (=עוברים בו פעם אחת למשך פרק זמן קצר מאוד – לא חוזרים אליו) לפני הפקודה לא היו משאבים, לאחריה כבר יש משאבים.
2. **Running**- מצב שנעים בו. תהליך שה-intructions שלו רץ ע"י המעבד (מוקצה עבורו משאב).
3. **Waiting**- מצב שנעים בו. תהליך שמחכה ל-event שיקרה (למשל פנייה לI/O ומחכה שהוא יתרחש, או בקשה לצאת לsleep והוא מחכה לinterrupt של הזמן שיסתיים)
4. **Ready**- מצב שנעים בו. תהליך שמחכה שהschedule ייתן לו זמן מעבד.
5. **Terminated**- התהליך מסיים את ריצתו. מצב "מומנטרי" גם כן. לפניו היו משאבים, אחריו אין.

<u>כמה תהליכים יכולים להיות במערכת בכל רגע נתון ובכל state?</u>

- הrunning קטן שווה למספר <u>הליבות</u>.
- ה new וה-terminated הם או 1 או קטנים שווים למספר <u>הליבות</u> (המעבדים) אלא אם כן במ"ה לא מרשים שיהיה יותר מאחד בו זמנית (למשל - למדנו עבור מ"ה שהן אסימטריות)
- ה- waiting וה-ready הם קטנים שווים למספר <u>התהליכים</u> (כי כל תהליך יכול להיות ברגע נתון באחד מהם, או כולם בו זמנית). או יותר מדויק – כמות התהליכים פחות כמות הליבות.



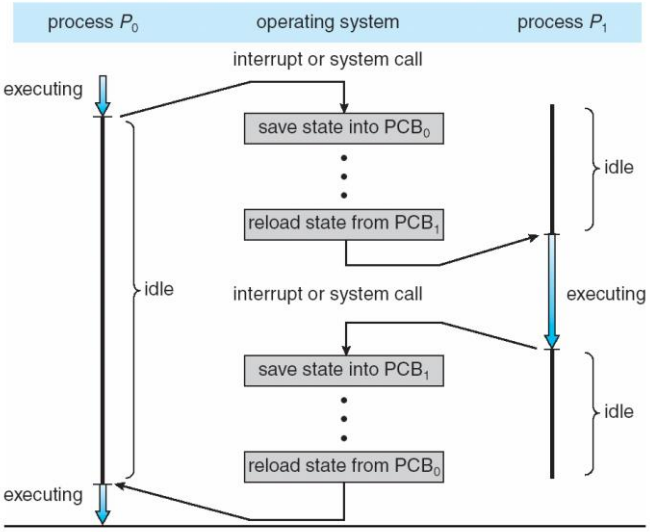
הסכימה בשקופית 10 – <u>מוסבר איך עוברים ממצב למצב</u>. כל תהליך מתחיל מ**new**. מרגע היצירה הוא עובר ל-admitted (נהיה תהליך ככל התהליכים – כמו קבלת תעודת זהות), והולך למצב **ready** (חלק מהpool של התהליכים במערכת)- ואז הוא יכול לרוץ והוא ממתין למעבד. למה שהוא לא יעבור על ההתחלה לwaiting? כי המעבד הוא המפתח לכל. לrunning לא נעבור על ההתחלה כי יש לחכות לscheduler שיפנה עבורנו מקום. כעת אנו בready, וממנו ניתן לעבור אך ורק ל**running**. למה לא נעבור לterminated? כי צריך להריץ exit, ואת זה רק נוכל לעשות בrunning. מrunning ניתן להגיע ל**terminated** (סיום הפקודות, exit וסיום), <u>או</u> ל**ready** (בעקבות interrupt או רצון של הscheduler

לקחת ממנו זמן מעבד) <u>או</u> ל**waiting** (יקרה כאשר ביקשנו בקשה לI/O, לא ניתן לרוץ ולכן אנחנו ממתינים).

<u>ההבדל בין waiting לready</u> הוא שב-waiting גם אם היו נותנים לי זמן מעבד לא הייתי יכול לרוץ, ואילו בready כן ניתן.

לכל תהליך נשמור <u>רשומה</u> שנקראת **PCB** (Process Control Block) ושם נשמור את כל המידע שצריך לדעת עבור התהליך - stateה של התהליך, את ה PC, את תוכן הרגיסטרים, מידע שקשור ל-scheduling של התהליך, מידע שקשור בזיכרון, מידע שקשור לaccounting, ומידע שקשור לסטטוס של I/O (נוכל לשמור שם את כל ההתקנים הפתוחים של התהליך , לא נרצה שתהליך אחר ייגש לאותו קובץ שאנחנו מחזיקים פתוח).

כשתהליך רץ, ונרצה להוריד אותו, ולהעלות תהליך אחר – זה נקרא **Context Switch** – כי אנחנו ממש שומרים את כל הContext של התהליך בזמן ההחלפה. איך זה פועל? הסבר בתרשים בשקופית 12 - יש לנו מעבד יחיד. ושני תהליכים שרצים במערכת p0, p1. כיוון שהמעבד הוא יחיד נוכל להריץ תהליך אחד ברגע נתון. p0 רץ, ויש interrupt או שהוא הריץ בקשה לI/O – מה עושים? מבצעים context switch – נשמור את ה- state שלו לתוך PCB0, המעבד בזמן שמירת הנתונים הוא idle והתהליך p0 עבר למצב waiting. כעת, נריץ את התהליך p1 ונטען את נתוניו למעבד. כשנרצה לחזור נטען את הcontext של p0 חזרה ונריץ אותו.



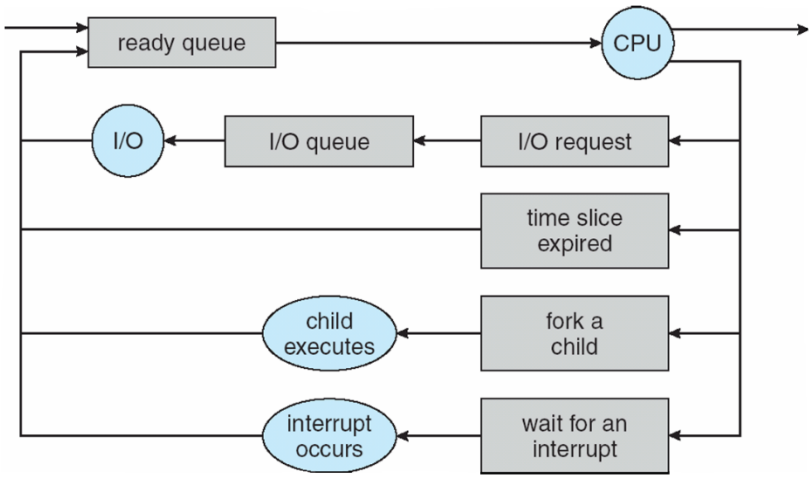
<u>הרעיון הוא</u> שבכל רגע נתון יהיה תהליך שיכול לרוץ על המעבד, כי נרצה <u>למקסם את ניצולת המעבד</u>. לא נרצה שיקרה מצב שתהליך יצא ל-I/O, ושאר התהליכים במצב waiting. למה? כי אז אין לנו אף תהליך שיכול להשתמש במעבד, והוא לא מנוצל כיאות.

איך ננהל את ניצול המעבד בצורה נכונה? נשתמש ב **Process Schedule** - אותה פונקציונאליות שיודעת להחליף תהליכים על גבי המעבד, ובו נשתמש בשלושה תורים:

1. **Job Queue** – כי הוא כולל את כל התהליכים במערכת.
2. **Ready Queue** – שם ישמרו כל התהליכים שהם גם בזיכרון וגם ממתינים להרצה – מוכנים להרצה ואין לנו צורך לטעון אותם כיוון שהם כבר יושבים בזיכרון (במצב ready). <u>האם יכול להיות קיים תהליך שהוא ready אבל לא בזיכרון?</u> כן. למשל נריץ הרבה תוכנות במקביל, ואנחנו במצב שאין לנו מספיק מקום להציג את כל תמונות הזיכרון של כל התהליכים האלו. לכן חלק ישבו בדיסק, ולא בזיכרון. כדי שירוצו – מישהו יצטרך לשלוף(לטעון) אותם מהדיסק לזיכרון ואז נוכל להריץ את הפקודות שלהם.
3. **Device Queues** – סט תהליכים שממתינים לI/O Devices. במקומות שהקריאות I/O הן קריאות אסינכרוניות-שולח בקשה וממשיך לרוץ הלאה- צריך לנהל מערכת של תורים כך שלכל התקן של I/O ננהל תור משלו של בקשות שממתינות לקבל ממנו שירות .

<u>גלגול חייו של התהליך בין התורים השונים (שקופית 15):</u>

בהתחלה התהליך מגיע לReady Queue, כי הוא במצב ready והוא רוצה להריץ את הפקודה הראשונה שלו, כלומר ממתין ל-scheduler. משם הוא יגיע למעבד (או לדיסק לפרק זמן מסוים ויחזור חזרה לזיכרון). משם הוא יסיים(Terminated) **<u>או</u>** ימצא את עצמו בI/O Queue (אם ביקש I/O) **<u>או</u>** שיחזור לReady Queue (זמן המעבד שלו הסתיים), **<u>או</u>** ימצא את עצמו בwaiting (למשל כתוצאה מיצירת בן, והמתנה עד שיסיים, ורק אז יוכל לחזור לready queue), **<u>או</u>** שעשה sleep עד שיקבל interrupt מהשעון שהוא יכול להמשיך לרוץ.



Scheduler- תפקידו להחליט מי מקבל זמן מעבד בכל רגע נתון, למרות שברוב מערכות ההפעלה יש שני Schedulerים:

1. **Long-term scheduler** – לטווח ארוך - מחליט איזה מהתהליכים יהיו בready queue, כלומר איזה מהתהליכים שבמצב ready יישמרו בזיכרון או בדיסק (יועברו לדיסק בעזרת swap out ובהמשך יחזרו לזיכרון בעזרת swap in). הוא ישות חשובה ומורכבת, ומופעל בתדירות נמוכה(כל כמה שניות או דקות), לכן ניתן למצוא אותו מורץ על גבי המעבד בפרקי זמן ארוכים. שולט על **הdegree of multiprogramming** (כמות התהליכים שירוצו במקביל על המערכת), ומשפיע על pool ים "מלאים" שעל-פיהם עובד **הShort-term scheduler**. ידאג

שבכל רגע נתון יהיה "מיקס טוב" של תהליכי I/O bound ותהליכי CPU bound (הסבר בהמשך), כדי שתהיה נצילות טובה של המעבד.

2. **Short-term scheduler** – לטווח קצר - מחליט לגבי מי מהתהליכים שיושבים ב ready queue יגיע לCPU כשהוא מסתיים, והוא מסלק תהליכים מהמעבד בשיקולים שנלמד בשיעור הבא. הוא צריך להיות זריז כי אחרת ייווצר overhead משמעותי, פשוט ומהיר, לאו דווקא חכם, כיוון שהוא תמיד מופעל (כל כמה מילי-שניות), וזמן הניצול שלו של המעבד הוא לא משמעותי. לכן הוא משתמש בשיטות שזמן עיבודן הוא קצר.

<u>אפיון התהליכים</u>:

1. **I/O bound** – תהליך שמבלה את רוב זמנו בבקשות I/O (למשל explorer שקורא מהמסך, מקבל inputים מהמשתמש וכו') – יש לו מעט מאוד פעולות חישוביות (יחסית קצרות).
2. **CPU bound** – תהליכים חישוביים (למשל תהליך שכל תפקידו לחשב פאי) – הם ממתינים למעבד, ומעט מאוד מבקשים לגשת לI/O.

המתזמן לטווח ארוך צריך לדאוג שבכל רגע נתון בזיכרון יהיה מיקס של 2 סוגי התהליכים הללו .

בחזרה ל- **<u>Context Switch:</u>**

זמן הביצוע של context switch הוא overhead (שעשוי להטריד אותנו), שכן המערכת איננה מבצעת עבודה שמשמשת את המשתמש. נרצה שהזמן הזה יהיה כמה שיותר מהיר, למשל בעזרת חומרה - מערכת Sun UltraSPARC היא מציעה מספר סטים פיזיים של רגיסטרים – ואז בContext switch במקום לשמור בPCB, נסתכל על סט רגיסטרים אחר, ובהמשך נחזור אל הראשון.

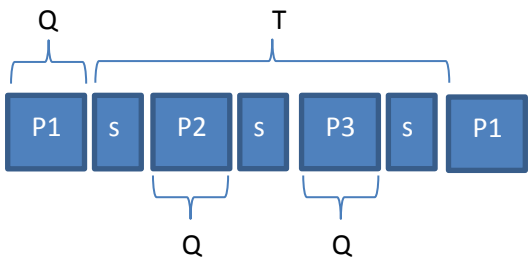
<u>שאלה 3</u> (10 נקודות)

במערכת מחשב N תהליכים החולקים CPU באמצעות מנגנון RR (N>=2). הנח כי כל context switch לוקח S מילישניות ושכל time quantum אורך Q מילישניות. לצורך פשטות, הנח כי התהליכים אף פעם לא עושים blocking (באף event) ורק עוברים בין ה-CPU וה- ready queue. מהו הערך המקסימלי של Q כך שהזמן המקסימלי בין הרצת שתי instructions של אותו תהליך הוא T מילישניות? (התשובה צריכה להיות פונקציה של N, S ו- T)

תשובה:

$$Q < \frac{T - N * S}{N - 1}$$

$$(n-1)Q + nS \le T \rightarrow Q < \frac{T - nS}{n - 1}$$



תהליך חדש מיוצר ע"י פקודת system call שמיועדת לכך. יש תהליך אב שמייצר תהליך בן שמייצר תהליך נכד וכן הלאה כך שנוצר עץ תהליכים.

ישנן 3 שיטות לחלוקת המשאבים בין האב לבן :

1. האב והבן יחלקו את כל המשאבים- כל מה ששייך לאב שייך גם לבן.
2. האב מחליט כמה משאבים מקבל הבן.
3. אין בכלל שיתוף ברמת המשאבים של האב והבן. הבן מקבל את כל המשאבים ממערכת ההפעלה.

הבן משכפל את תמונת הזיכרון של האב בעת יצירתו. ה- PC שלו נמצא באותו מקום- בדיוק אחרי פקודת ה**fork()** - נוצר address space תואם לזה של האב. הפקודה **exec()** - טוענת לתוך ה process תכנית חדשה.

הערה - לקוד בשקופית 24 - כיוון שמריצים תכנית חדשה בבן – אז לא נגיע לחלק האחרון באב- כי אנחנו מריצים כבר קוד אחר.

האבא מחכה לבן ע"י ביצוע **wait()**. בסיום הרצת התהליך הוא מריץ **exit()** ומערכת ההפעלה מוחקת אותו מרשימת התהליכים ולוקחת את משאביו ושמה ב-pool שלה. אם התהליך הוא תהליך בן אז מועבר הoutput לתהליך האב.

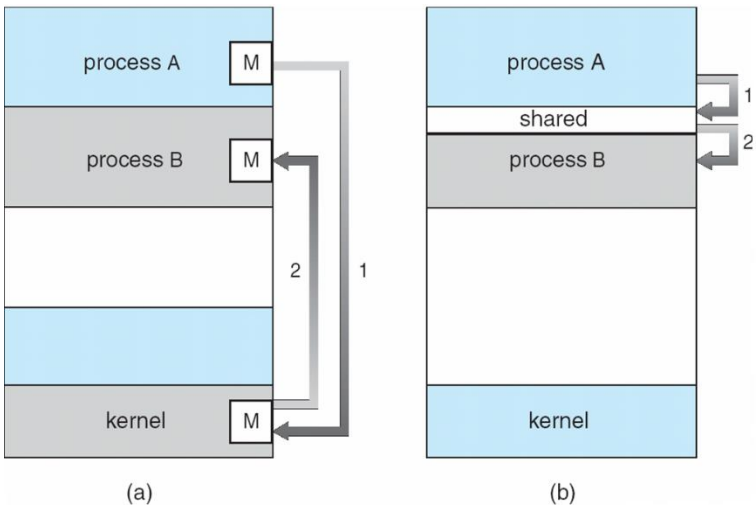
תהליך האב יכול לגרום להפסקת הבן במקרים הבאים – **abort()**:

- תהליך הבן משתמש ביותר משאבים ממה שהוקצו לו.
- המשימה שלשמה נוצר תהליך הבן הסתיימה או שאינה נדרשת עוד – הבן שואל את האב האם צריך להמשיך לרוץ.
- תהליך האב עצמו סיים את פעולתו - בחלק ממערכות ההפעלה לא ניתן להשאיר תהליך רץ כאשר תהליך האב מסתיים – במקרה כזה עושים **cascading termination**.

יש אופציה שבה האב פשוט עוצר את הבן.

2 מודלים של IPC - תקשורת בין תהליכים:

1. **shared memory** - זיכרון משותף- מאפשר לשני תהליכים לגשת לאותו זיכרון פיזי. נרצה שהוא יהיה משויך לאחד מה-processים. נסתכל על ה-process שיצר אותו ונאפשר לאחרים לגשת לזיכרון הזה.
2. **Message passing** – זיכרון לכל תהליך + מנגנון העברת הודעות ומערכת ההפעלה היא זו שתומכת לנו במנגנון העברת ההודעות הללו.



הימני- זיכרון משותף, השמאלי – מנגנון העברת הודעות.

יתרונות השיטה הראשונה - יותר מהיר- כותב וקורא לזיכרון אין דרך יותר מהירה מזו.

יתרונות השיטה השנייה - יותר בטוח- מישהו מנהל אותו. יותר קל לניהול.

Shared memory :

Producer-Consumer - שני תהליכים שצריכים לתקשר ביניהם- יש לנו יצרן שמייצר משהו וצרכן שצורך את הייצור הזה. לדוגמה – הקומפיילר(יצרן) מייצר קוד אסמבלי והאסמבלר(צרכן) צורך את הקוד הזה.

Unbounded buffer- ייצור כמה תהליכים שרוצים.

Bounded buffer- מניחים שה-buffer סופי ולכן לא נייצר בלי סוף תהליכים. (צריך ניהול יותר הדוק).

דוגמת קוד בשקופית 30 במצגת.

Message passing :

מהווה מנגנון לתקשורת וסנכרון בין תהליכים. לא יהיה פה שום דבר משותף. מ"ה עושה את זה בשבילנו. IPC – עושה שתי פעולות עיקריות: send ו-receive.

אם יש שני תהליכים שרוצים לתקשר ביניהם-הם יצרו ערוץ תקשורת ויעבירו הודעות ביניהם. הlink יתאפשר באמצעות חומרה.

תקשורת ישירה- שליחת הודעות עם מידע על מי שלח/שולח. מחייב את מ"ה לתמוך ביצירת ערוצי תקשורת בצורה אוטומטית. ערוץ שנוצר הוא ערוץ בין process ל- process. בין כל שני processים יהיה לנו ערוץ תקשורת אחד בלבד ונוכל להעביר הודעות לשני הכיוונים.

חסרון- שינוי המזהה של התהליך- תוקע את התקשורת, כי התהליכים האחרים מכירים את התהליך עם המזהה הישן שלו. צריך לדאוג לא לאפשר שינוי מזהה של תהליך בזמן תקשורת. שיטה פחות אפקטיבית בגלל המגבלה.

תקשורת לא ישירה- ננהל Mail box ע"י מ"ה והתהליכים השונים ירשמו לתיבות הדואר. הצורה היא send ו- receive כאשר A היא תיבת הדואר המשותפת. התיבה יכולה להיות מקושרת במספר רב של תהליכים וכל תהליך יכול לתקשר דרך כמה תיבות דואר. ניצור תיבות ונשמיד תיבות (פעולות קיימות).

בעיה- מי מקבל את ההודעה- מי ניגש לתיבת הדואר?

1. לא נאפשר לתיבה להיות מקושרת ליותר משני תהליכים – תקשורת ישירה.

2. רק תהליך אחד יוכל לעשות receive בזמן נתון.

3. המערכת מאפשרת לבחור אקראית מי המקבל. המערכת תבחר אלגוריתם מסוים שלפיו נחליט מי המקבל.

Synchronization :

מנגנון ההודעות יכול להיות blocking (סינכרוני) או non-blocking (לא סנכרוני).

סינכרוני- השולח מפסיק את פעולתו עד שהמקבל מקבל את ההודעה. המקבל עוצר את עצמו עד שההודעה זמינה לקבלה.

לא סינכרוני- השולח שולח את ההודעה וממשיך בפעולתו. המקבל מקבל את ההודעה גם אם היא null או invalid.

Buffering :

ניתן להשתמש באחת מ-3 אפשרויות:

1. **Zero capacity**- גודל הבאפר אפס. כלומר אין טווח שישמור את ההודעה עד שהצד השני יקבל אותה. הצד השני חייב לקבל אותה אחרת לא נוכל להמשיך כי אין איפה לאחסן אותה.
2. **Bounded capacity**- יש גודל סופי לבאפר. רק כשיתמלא נעצור ונעבוד בצורה סינכרונית.
3. **Unbounded capacity**- נתעלם מהבאפר ונניח שיש לנו גודל אינסופי. כך נוכל לשלוח הודעות בלי לחכות.

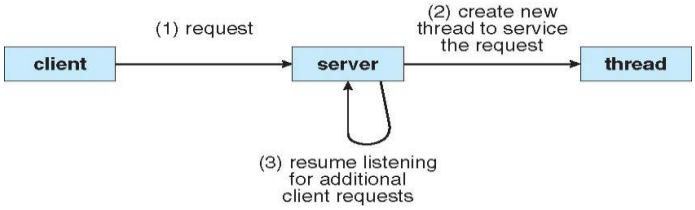
Client-Server - מנגנונים:

1. **Sockets** - נקודת קצה לתקשורת.
2. **RPC** - תהליך רוצה לשלוח בקשה פרוצדורלית לתהליך אחר, שבה יש פונקציה שנוכל להפעיל. נייצר שכבה ברמת הקליינט והשרת. בצד הקליינט השכבה תייצר את הבקשה ותעטוף אותה כך שבצד השרת השכבה תדע לפתוח את ההודעה עם השאילתה הפרוצדוראלית כאילו הקליינט ישב בשרת וביצע את זה בצורה ישירה.
3. **RMI**- בג'אווה, שמבקשים מתוכנית בגאווה שרצה בVM אחת שתפנה לאובייקט ב-VM אחר כאילו שהם רצו על אותו ה-VM.

הרצאה 5 :

עד כה הנחנו כי כל תהליך מריץ תכנית באמצעות thread יחיד. כל תהליך יכול להריץ רק דבר אחד במקביל. רוב האפליקציות משתמשות ב- **multithreading**.

השרת מייצר תהליך חדש בזמן האזנה והוא ייתן שירות לבקשה –fork(). אופציה אחרת יהיה לייצר thread – בשיטת multithreading. שיטה זו מהירה יותר, ויש לה יותר יתרונות מהשיטה השנייה.



threadים יכולים לרוץ במקביל. משימות רבות של התהליך יכולות להיות ממומשות כthreadים עצמאיים: עדכון תצוגה, הבאת מידע, spell check, שירות בקשות שמגיעות דרך הרשת.

מדוע נרצה להשתמש בthread ים במקום בתהליכים?

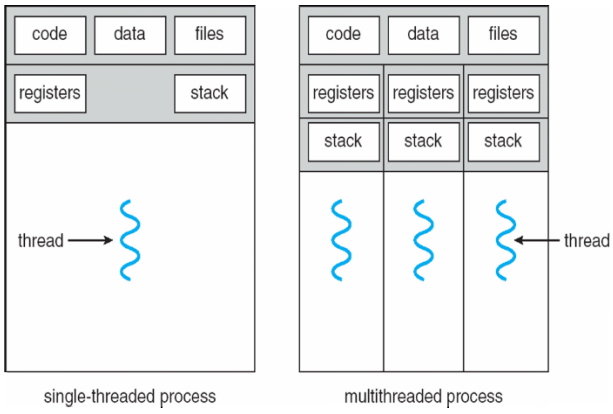
- הרבה יותר זול מבחינת overhead.
- שיתוף אינפורמציה. ניתן לשתף מידע ביותר קלות מאשר בתהליכים.
- הkernel משתמש בmultithreading .
- יצירת תהליך חדש הרבה יותר מורכבת ובזבזנית מיצירה של thread.

הthread הוא יחידה בסיסית של ניצול מעבד. מדוע רק למעבד? - כיוון שלdevice יש תור עדיפויות מובנה ואין צורך בכך. רק במעבד אין תור כזה- שעושה סדר בבקשות. תהיה בעיה ברגע שתהליכים יכנסו לblock.

הthread מורכב מ- ID, PC, Register set ו-Stack.

הthread חולק עם שאר הthreadים את החלקים הבאים: code section, Data section, other OS resources such as open files.

למה אין לthread heap משלו? המנגנון הזה של ה- heap הוא מנגנון יותר קל לניהול- זו הקצאה דינמית שממשיכה ומלווה אותה לאורך כל התהליך, בניגוד למחסנית- שקשה לנהל אותה כשיש כמה thread ים שרצים במקביל ולכן לכל thread מחסנית משלו.



תרשים של תהליך בודד עם thread בודד (בשמאל) ותהליך שמכיל מספר thread ים.(בימין)

יתרונות:

Responsiveness - מאפשר לתכנית להמשיך לרוץ גם כאשר חלק ממנה הוא חסום או שמבצע פעולה ארוכה.

Resource Sharing - הthread ים חולקים את אותו זיכרון ומשאבים של התהליך. כך יש אפשרות להעביר את אותו מידע בין הthread ים. התקשורת ביניהם יותר מהירה ופשוטה.

Economy - יותר כלכלי ליצור thread ים ולבצע context switch לthread ים (ביחס לתהליך) מכיוון שהם חולקים את משאבי התהליך - לthread ים יש יותר מרכיבים משותפים.

Scalability - אותו תהליך יכול לרוץ על מספר מעבדים במקביל באמצעות שימוש בthread ים.

היתרון הראשון לא קשור לכמות ה-cores. היתרון הרביעי- מתמקד במספר המעבדים. זה שני יתרונות שונים לכך שניתן לחלק thread ים.

הראשון מגביר את היכולת שלנו לייצר יישום אינטראקטיבית. למשל ניצור thread שימשיך לייצר אינטראקציה עם המשתמש למרות שמטפלים בפעולה ארוכה.

הרביעי מייצר לנו יכולת להריץ דברים מהר יותר.

Multicore Programming:

אתגרים חדשים בתכנות למערכת Multicore או multiprocessor:

- מציאת תחומים פעילויות שניתן לחלקם בצורה נקייה למשימות שיכולות להתקדם במקביל. כלומר נרצה שתהיה הפרדה נקייה בין הפונקציונאליות של הthreadים.
- תכנון המשימות המקביליות – נרצה לחלק משימות ל- thread באופן שווה (balancing). ז"א חלוקה לthreadים שיש להם תכולה יחסית דומה).
- חלוקת ה data ויצירת מבני נתונים מתאימים לתמיכה בהרצה במקביל- צריך לדאוג לסינכרוניזציה.
- קושי משמעותי ב- debugging ו testing לעומת טיפול ב- thread בודד. למשל יש לנו בעיה לשחזר תקלה כי לא יודעים מי רץ קודם – אין לנו שליטה על מי ירוץ קודם.

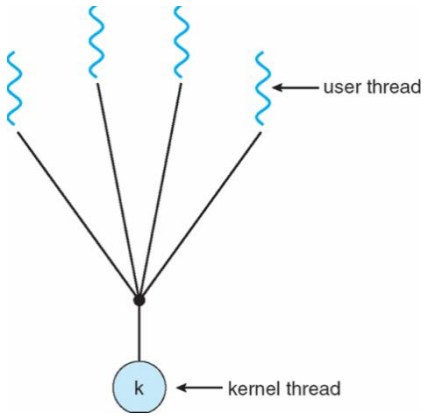
User and Kernel Threads:

1. **User thread** – threadים שמנוהלים ב-user space ע"י ספרייה כלשהי שלמעשה קובעת ברמת ה-user space איזה קוד ירוץ בכל רגע ורגע. כל הthreadים הללו שייכים לתהליך שייצר אותם. ה- kernel לא מודע אליהם.
2. **Kernel threads**– threadים שנוצרים ומשובצים על ידי הkernel. הם יקרים משמעותית מהuser thread.

המעברים בתוך ה- user space הרבה יותר פשוטים. לא צריך לעשות context switch, אלא בסה"כ להזיז ת ה-PC למקום אחר בקוד של ה thread האחר.

דוגמאות למערכות נתמכות ע"י kernel threads – וינדוס, לינוקס, SOLARIS, MAC.

דוגמאות ל-3 ספריות עיקריות של user thraeds- POSIX Pthreads, Windows threads , Java threads.



4 שיטות לקישור בין שני הסוגים:

Many to one- שיבוץ על kernel thread אחד מספר threadים של user. כל הניהול נעשה בספריה היושבת בuser space . יעיל משיקולי overhead כיוון שהמעבד לא עושה context switch הוא מבחינתו מריץ thread אחד.

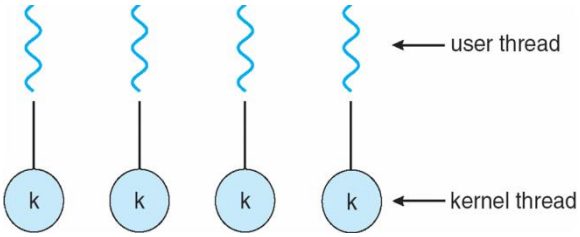
חסרונות: איבדנו את היתרון בשימוש במולטי threadים. התהליך כולו ייכנס למצב blocked אם אחד מהthreadים קורא ל system call שהוא blocking. אין כאן באמת הרצה במקביל כאשר יש למעבד מספר ליבות(או מספר מעבדים), מתייחס אליו כאל thread בודד.

יתרון: יותר מהיר ויותר יעיל כי כל ההחלפות מיוצרות ברמת הuser space .

one to one- כל user-level thread ממופה ל- kernel thread משלו (בעת יצירת user level thread נוצר עבורו kernel thread).

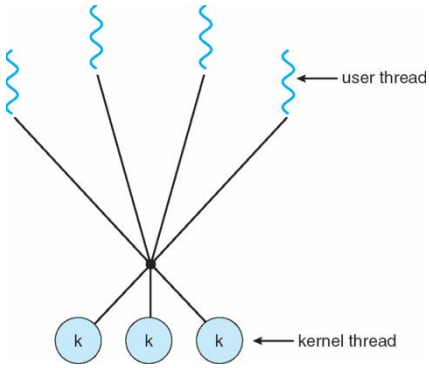
יתרונות: מקביליות מקסימלית בהרצה, אין מצב שבו blocking thread שמחכה לi/o למשל- חוסם את כל יתר ה- threads- הם ימשיכו לרוץ.

חסרונות: מחייב יצירה של המון kernel threads , ה-overhead של ניהול כל ה- kernel threads מביא בהרבה מערכות למגבלה על כמות ה –threads המקסימלית.



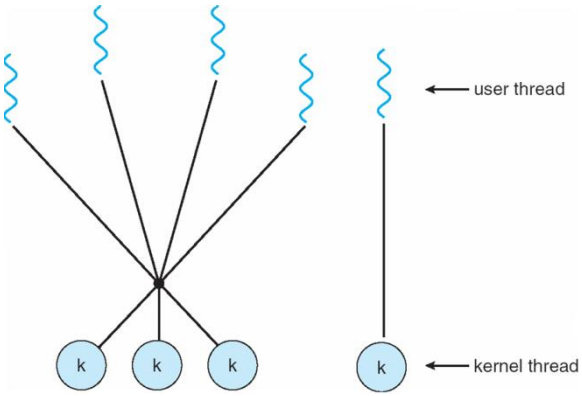
many to many - מאפשר למספר user level threads להיות מקושרים למספר kernel threads. מאפשר למערכת ההפעלה שליטה על כמות ה- kernel threads שיפעלו במערכת.

יתרון: שמצד אחד לא מנפחים את מ"ה עם המון kernel threads ומצד שני לא עוצרים את כל ה processים ברגע שאחד מ user thread ביצע פעולות שעוצרות כגון פעולת i/o.



two-level model- כמו ה- many to many אבל מאפשר ל- user thread להוות מגבלה עבור kernel thread.

יתרונות: אם יש לנו user thread קריטי- נשדך לו kernel thread כך שלא יפגע בהעדר kernel thread. נהנים משני העולמות.



Threads libraries- הממשק שלנו ליצירה ולניהול threadים. בד"כ יש ספרייה בuser space . מבחינת מימוש יהיו מ"ה שיתנו תמיכה מרמת הuser ועד רמת הkernel. נלמד את ה- API של הספריה pthreads. (ילמד בתרגול) .

threadים ב-java- מנוהלים ע"י ה-jvm. יש 2 אופציות : מימוש הממשק runnable או לקחת thread class קיים ולדרוס את הפונקציה run שלו.

ניתן להרחיב את המחלקות הממומשות כבר, ניתן גם לממש interface.

קריאה לפונקציה ()start יוצרת זיכרון ומקצה thread חדש ב-jvm. הקריאה לפונקציה ()run מתחילה לבצע את הפעולה הרצויה.

סוגיות עבור Threading :

מה קורה כאשר עושים ()fork ומיד אחריו- ()exec? האם צריך לשכפל את כל הthreadים באותו תהליך או לשכפל את הthread היחיד שממנו עשיתי את ה ()fork?

תלוי –אנו יכולים לפעול בשתי השיטות – תלוי בתוכנית שאנחנו מריצים. נשים לב כי על ידי הרצת הפעולה ()exec, הרי אנו יוצרים את כל מרכיבי הThread מחדש. לכן, אין סיבה שנעתיק את כל הThreadים של התהליך אם גם ככה הם ימחקו בגלל פקודת ה()exec, כלומר נעתיק רק את ה thread שממנו עשינו ה()fork.

Thread Cancellation :

נרצה לעצור איזשהו thread ולהוריד אותו מה thread pool שלנו. יש 2 שיטות:

1. **אסינכרונית-** הורגים את הthread וייתכן שהוא לא סיים את מה שהיה צריך לעשות. שיטה זו מאוד מהירה, אבל ייתכן שנסתבך עם משאבים שאמורים להתפנות על ידי הthread שלנו. לא מסיים לשחרר זיכרון לדוגמה.

2. **deferred-** מסמנים לthread שהוא צריך להרוג את עצמו ואז הוא עושה הכנות אחרונות והורג את עצמו. הthread בודק איזשהו פלאג שאומר לו אם הוא צריך להמשיך לחיות. אם לא הוא הולך לאיזושהי פונקציה שעושה דברים לפני יציאה ומסיים.

Signal handling :

בUnix- המערכת מעבירה סיגנל לתהליך שמסמן לו שמשהו קרה.

למי מבין הthreadים נעביר את הסיגנל?

1.להעביר לthread שהסיגנל רלוונטי לגביו.

2.להעביר את הסיגנל לthread היוצר.

3. להעביר את הסיגנל לכל הthreadים שבאותו תהליך.

4. להעביר את הסיגנל לתת קבוצה של threadים שאמורים לקבל סיגנלים שהגדיר מנהל התוכנית.

5. להעביר את הסיגנל לthread הספציפי שנגדיר שיקבל את כל הסיגנלים של התהליך וידע לטפל בהם.

: Thread pools

מייצרים pool של threadים מראש וברגע שיש משימה שדורשת טיפול- נקצה Thread מתוך הpool. חסכון בזמן יצירה, ונותן שליטה מלאה על מקסימום הthreadים שיהיו במערכת. (שכן יצירה בלתי מוגבלת של threadים עלולה לסיים את משאבי המחשב כמו CPU או זיכרון).

: CPU Scheduling

התהליך שלנו מבלה את חייו במעבר בין **CPU burst** ל**I/O burst** – בין הרצה רציפה של פקודות ליציאה ל- I/O. כל תהליך יתחיל בהרצת הפקודה הראשונה - CPU burst ויסתיים בהרצה של exit - CPU burst**.** במצגת יש סכמה שמראה את התפלגות ה-CPU burst במחשב. בסכמה במצגת ניתן לראות שיש מעט מאוד burstים קצרים והרבה burstים ארוכים.

Multi-programming – חלוקת עבודת המעבד למספר תהליכים בצורה יעילה.

תפקיד הscheduler הוא לבחור מי התהליך שירוץ ברגע נתון. בכל פעם שמתפנה מעבד נרצה שהscheduler ינתב אליו תהליך שמחכה בתור.

ישנם 4 מצבים אפשריים בהם הscheduler יכנס לפעולה :

- תהליך עובר ממצב running למצב waiting – המעבד מתפנה ואפשר להעלות אליו תהליך חדש.
- תהליך עובר ממצב running למצב ready.
- תהליך עובר ממצב waiting לready – יש מישהו נוסף שמחכה לקבל זמן מעבד.
- תהליך מסיים את פעולתו.

בהתאם, יש לנו 2 מצבים של הscheduler :

- **Preemptive** – פועל בכל אחד מ-4 המצבים הנ"ל. מסוגל לעזוב תהליך באמצע ריצתו לטובת תהליך חשוב יותר (I\O).
- **Nonpreemptive** – פועל רק במצבים 1 ו-4. הוא אינו מסוגל להפסיק תהליך באמצע וללכת לתהליך אחר.

Dispatcher – הוא מבצע את ההחלפות בפועל. הזמן בו לוקח לו לבצע את ההחלטה נקרא **dispatch latency** – זהו זמן overhead שיכול להיות מאוד ארוך.

מהם המדדים שלפיהם נוכל להעריך את הscheduler?

- **ניצולת ה CPU**– מודדת כמה אחוז מהזמן המעבד עבד. אם הscheduler הוא טוב אזי הוא ידאג שהמעבד כל הזמן יריץ על גביו משהו – מקסימלית.
- **תפוקה** – מספר התהליכים שסיימו את פעולתם פר יחידת זמן (למשל - 20 תהליכים בשעה) – מקסימלית.
- **Turnaround time** - הזמן שעבר מאז שהתהליך נוצר ועד שהוא סיים את פעולתו - מינימלית –

מורכב מ:

- מהזמן שבו התהליך ממתין להיכנס לזיכרון בזמן יצירתו או בכל זמן אחר במהלך פעולתו.

- מהזמן שבו התהליך ממתין בready queue.

- מהזמן שהתהליך רץ על גבי המעבד.

- מהזמן שבו הוא ממתין ל-I\O.

- **Waiting time** - מתוך כל ארבעת הזמנים הנ"ל המרכיב החשוב והמשפיע ביותר הוא זמן ההמתנה לזמן מעבד - הזמן שבו התהליך ממתין בready queue – מינימלית.
- **Response time** - הזמן שעבר מרגע שהתהליך ביקש בקשה מהמעבד ועד שהוא קיבל התייחסות ראשונה מהמעבד – מינימלית.

<u>שיטות ל-scheduling:</u>

- **FCFS- first come first served** – התהליכים מטופלים לפי <u>סדר כניסתם</u> לתור (שימו לב – לא לפי סדר הגעתם כי כמובן שמספר תהליכים יכולים להגיע יחד אך יכנסו לתור בסדר שונה).
  שיטה זו היא nonpreemptive – אינו עוצר באמצע , מסיים את הרצת התהליך הנוכחי ועובר לתהליך הבא. שיטה זו מאוד מושפעת מסדר הגעת התהליכים – אפקט השיירה. בדוגמה הראשונה התהליך הכי ארוך הגיע ראשון ולכן עיכב את שאר התהליכים בתור מה שגרם לwaiting time פר תהליך להיות גדול יותר. בדוגמה השנייה בה התהליך הארוך ביותר הגיע אחרון ה-waiting time קטן.
- **SJF – shortest job first/shortest burst first** – מתסכל בתור התהליכים ובכל פעם בוחר את התהליך שה CPU burst שלו הוא הקצר ביותר. ניתן לממש ב-2 שיטות :
  - Nonpreemptive – מתחיל ומסיים עם אותו התהליך.
  - Preemptive (shortest remaining time first) – עבור כל תהליך חדש בוחן את התור ובודק האם התהליך שהגיע זה עתה, זמן הריצה הנותר(השארית) שלו הוא הקצר ביותר ובהתאם מבצע החלפה. מה שמטריד בשיטה זו היא שתמיד יכולים להגיע תהליכים קמרים יותר שלא יאפשרו לתהליכים אחרים להתקדם בתור.
  שיטה זו תמיד תהיה טובה יותר להשגת average waiting time מינימלי, וכן ה-Preemptive בהכרח תמיד יהיה טוב יותר מאשר ה-Nonpreemptive . נוכל להראות זאת ע"י הוכחה שכל החלפה של תהליך ארוך בתהליך קצר ימזער את ה- average waiting time באינדוקציה.
  הבעיה בשיטה זו היא שעלינו לדעת את הburst time של כל תהליך וזאת ע"י חיזוי. במצגת ישנה שקופית ובה שיטה לחיזוי. התחזית תהיה בנויה על כל הburstים שראינו עד עכשיו מהburst הראשון, כאשר המשקל שאנו נותנים לתצפיות שראינו הולך ודועך ככל שהן יותר רחוקות מהזמן הנוכחי. נקבע את אלפא לפי התהליך שלנו – לתהליכים שהם מאוד דינאמיים נשתמש באלפא גבוהה ולתהליכים שהם יותר סטטיים נשתמש באלפא יחסית נמוכה.

1. t_n = actual length of nth CPU burst
2. τ_{n+1} = predicted value for the next CPU burst
3. $\alpha, 0 \le \alpha \le 1$
4. Define: $\tau_{n+1} = \alpha\, t_n + (1-\alpha)\tau_n$.

עבור תהליכים דינאמיים – נרצה לתת דגש ל-tn, כלומר נרצה להתבסס על ה burst time האחרון האמתי שהיה לנו. למה? – מכיוון שהתהליך דינאמי, קשה לחזות מה יהיה הburst time הבא שלו. לכן, נבחר אלפא גדולה ככל שניתן כך ש-tn יהיה כמה שיותר

גדול, והחיזוי יוכפל במשלים שלו(הסתברות) ויהיה כמה שיותר קטן. בתהליכים סטטים נעשה בדיוק להיפך.

- **Priority Scheduling** – לכל תהליך יהיה מספר שיגדיר את העדיפות שלו, ה- CPU יטפל בכל פעם בתהליך שהעדיפות שלו היא הגבוהה ביותר מתוך הready queue. אנחנו נבחר איך להשתמש במספרים האלה – מספרים נמוכים – עדיפות גבוהה\ מספרים גבוהים – עדיפות גבוהה. נרצה להשתמש בשיטה הראשונה שבה מספרים נמוכים מייצגים עדיפות גבוהה כי כך נצטרך לקרוא פחות ביטים. גם פה יש לנו preemptive ו- nonpreemptive.
הבעיה העיקרית בו היא "הרעבה" – תמיד יכול להגיע מישהו שהעדיפות שלו יותר טובה ובכך לדחוק תהליכים אחרים הצידה ולגרום להם לא לקבל זמן מעבד (ב-dead lock אנחנו מחכים למשהו שלעולם לא יקרה אם לא תהיה התערבות חיצונית). הפתרון לבעיה הוא aging – תהליך שיושב כבר הרבה זמן במערכת בגלל העדיפות שלו, נגדיל לו את העדיפות.
- **Round robin** – שיטה זו נותנת לכל תהליך הזדמנות. מצוינת למערכות מרובות משתמשים שדורשות זמן תגובה יחסית מהיר. כל תהליך יקבל זמן יחסית קצר על גבי המעבד שאנחנו נחליט אותו כזמן קבוע לכל התהליכים– **time quantum** וברגע שהוא ניצל זמן זה ההרצה שלו מופסקת והוא מועבר לסוף התור. אם התהליך סיים ולא ניצל את הזמן שניתן לו, נוריד אותו מהתור ולא נכריח אותו להישאר על המעבד עד תום הזמן. נעבור על התור שלנו בצורה ציקלית.
אם הtime quantum שניתן יהיה יחסית גדול אזי נגיע לביצועים כמו של FCFS.
אם הtime quantum שניתן יהיה יחסית קטן אז התהליכים יסיימו כולם יחסית מאוחר.
שיטה זו היא בהגדרתה preemptive.
כמובן שהtime quantum מאוד משפיע על זמן הcontext switch – ככל שזמן הtime quantum קטן יותר -> יש לנו יותר context switchים.
נבצע context switch בכל פעם שתהליך עולה על המעבד ובכל פעם שהוא יורד ממנו.
Turnaround time – הזמן החולף מהרגע שהתהליך עולה על המעבד עד שהוא מסיים את ה burst שלו.

בסכמה ניתן לראות שאין קשר ישיר בין הquantum time ל-turnaround time – הם אינם גדלים או קטנים יחד. אבל עבור q=6 ה-average נשאר סטטי כי זה הופך להיות FCFS כגודל התהליך הארוך ביותר.

Multilevel queue – ניקח את הready queue ונפרק אותו למספר תורים נפרדים. דוגמה לחלוקה – בתור הראשון יהיו תהליכים שאכפת לנו מזמן התגובה שלהם ובתור השני כאלה שלא אכפת לנו מזמן התגובה שלהם. כעת על כל תור נוכל ליישם שיטת scheduling אחרת.
נוכל לשפר אותו ולהפוך אותו ל-**Multilevel feedback queue** – העברת תהליך מתור אחד לאחר. התור האחרון תמיד יעבוד בשיטת FCFS.

עלינו להגדיר :

- כמה תורים יהיו לנו.
- לכל תור עלינו לקבוע באיזו שיטת scheduling נשתמש.
- מה השיטה שבה נשדרג תהליכים – איך נעביר אותם מתור נמוך לתור גבוה.
- להפך – איך נעביר תהליכים מתור גבוה לתור נמוך.
- עבור תהליך חדש שמגיע – לפי איזה כלל נחליט לאיזה תור הוא יכנס.

בדוגמה – בתור הראשון כל תהליך מקבל 8 מילישניות ומיד לאחר מכן עובר לתור השני, כלומר כל תהליך שאורך יותר מ-8 מילישניות יעברו לתור השני (שיש לו Priority יותר נמוך). בתור השני כל תהליך יקבל 16 מילישניות על המעבד ובמידה והוא דורש זמן נוסף (גדול מ-24 מילישניות –כי 8 +

16) הוא יעבור לתור האחרון שפועל לפי FCFS. תהליכים בתור ה-3 הם תהליכים שצורכים המון זמן מעבד. במודל זה כל תור מקבל זמן CPU רק כאשר התור שלפניו מתרוקן.

היתרון בדוגמה הזו – תהליכים קצרים יסתיימו יחסית מהר לאחר התור הראשון, תהליכים ארוכים יותר יעברו לתור הבא בתור ושם יקבלו burst נוסף כאשר בתור הראשון הם יקבלו זמן מעבד מהיר, בתור השני זמן מעבד פחות מהיר אבל עדיין יחסית מהיר וכן הלאה.

מה קורה כשיש לנו כמה מעבדים? – לכאורה יש לנו כמה מעבדים שצריכים להסתכל על אותו ready queue ואז יכולה להיווצר בעיית סינכרוניזציה – כולם ינסו לשבץ על עצמם את אותו התהליך. מצד שני אם היינו מנהלים תור אחד שהיה משרת את כל המעבדים זה היה מאט את פעולת ההקצאה כי המעבדים לא היו עובדים בו זמנית. ישנן 2 אפשרויות במקרה זה :

- **אסימטרי** - קיים רק תור אחד לשני המעבדים כאשר רק מעבד אחד ינהל את התור ומתזמן את התהליכים.
- **סימטרי** - כל מעבד יתזמן לעצמו עצמאית את הready queue שלו. יהיו לנו תורים נפרדים וכך תהליך יכול להימצא בתור אחד בו זמנית.

Processor affinity – נרצה שתהיה זיקה בין התהליך לבין המעבד שעליו הוא רץ כרגע. לשם כך ניתן להתנהל ב-2 שיטות :

- **Soft affinity** – המלצה שהתהליך יישאר על אותו מעבד - לא מחויב לקרות בפועל.
- **Hard affinity** – התהליך מחויב לרוץ תמיד על אותו המעבד ולכן אם התפנה מקום במעבד השני שאינו שייך לו התהליך ימשיך לחכות לזמן על גבי המעבד שיש לו זיקה אליו.

מערכת real time – מערכת שבה הדברים צריכים לקרות בזמן מאוד מאוד ספציפי כתגובה למשהו שקורה. כל השיטות שלמדנו הן קצת בעיתיות בהקשר של מערכות אלו.

מערכות **Hard real-time** – מאפשרות להשלים משימה קריטית בתוך פרק זמן מוגדר מראש:

- ה- process מבקש לקבל זמן מעבד עד ל- deadline מסוים.
- ה- scheduler מבטיח את קיום הבקשה (resource reservation) או דוחה אותה.

מערכות **Soft real-time computing** – תהליכים קריטיים יקבלו עדיפות על-פני תהליכים "רגילים".

במערכות שהן **hard real time** – מאפשרות להשלים משימה קריטית בפרק זמן מוגדר מראש.

אלגוריתם scheduling מותאם מערכת – 4 שיטות שעוזרות לנו להעריך אלגוריתמים שונים :

- **Deterministic modeling** – לוקחים סט של קלטים דטרמיניסטיים ומחשבים עבור כל קלט מה יהיה הwaiting time, turnaround time ומדדים אחרים ואז נבחר את הטוב ביותר לפי המדדים שמעניינים אותנו. היתרון שלה שהיא פשוטה לביצוע. הבעיה יכולה להיות שסט הקלטים שנשתמש בהם לא ייצגו מספיק טוב את מה שיכול לקרות במציאות.
- תורת התורים – **queuing models** – זהו תחום מחקר שמשתמש בנוסחאות לחישוב waiting time ומדדים אחרים לאובייקטים בתור.
- **Simulation** – נסמלץ תהליכים שמגיעים וניתן להם שירות לפי האלגוריתם אותו אנחנו בוחנים.
- **Implementation** – בכל פעם נשתמש באלגוריתם אחר על המערכת האמתית ובסוף נראה מה יצא הכי טוב. היתרון – הכי אמתי שאפשר. החיסרון – לוקח יותר זמן, גורם לנו לעבוד על סביבת אמת ולבצע עליה בדיקות – פחות טוב.

3 דוגמאות לscheduling במערכות הפעלה שונות :

- **Solaris** – מבוסס על time sharing – נותן לתהליכים time quantum לרוץ בו ועוצר אותם גם אם הם לא סיימו. תהליכים שיש להם עדיפות גבוהה יותר יקבלו time quantum קצר יותר כי אנחנו חולקים את התהליך הזה עם תהליכים אחרים באותה רמת עדיפות ולכן זה "פיצוי" על כך. **בטבלה** - כל תהליך יקבל עדיפות ובהתאם לעדיפות יקבל time quantum. ככל שהעדיפות גבוהה יותר-> time quantum יותר קצר. Time quantum expired – אומר מה תהיה העדיפות החדשה של התהליך במידה והוא סיים את הtime quantum אך לא סיים את הburst שלו, ברגע שהתהליך יחזור לתור נרצה להוריד את העדיפות שלו כי הוא כבר קיבל זמן מעבד ולכן הפעם הבאה שהוא יקבל זמן מעבד תהיה קצר יותר ארוכה. הטור האחרון – מה תהיה העדיפות של התהליך שהוא חוזר מו\O – מקבל עדיפות גבוהה יותר כי כנראה שכרגע יש לו משהו חשוב לעשות עכשיו עם הו\ O שקיבל. בטבלה יש לנו 60 רמות של עדיפות אך במערכת ההפעלה עצמה יש 110 רמות נוספות לתהליכים של מערכת ההפעלה עצמה כמו – system threads, real-time threads, interrupt threads. ממוקמים בתור לפי העדיפות שלהם.
- **Windows** – המימוש הקלאסי אומר לתת לכל תהליך 2 מאפיינים – העדיפות העיקרית, תת-עדיפות בתוך העדיפות העיקרית. כל עמודה בטבלה היא סיווג עיקרי של עדיפות וכל שורה בטבלה היא סיווג של תת-עדיפות.
- **Linux** – ככל שהעדיפות היא מספר יותר נמוך אזי העדיפות גבוהה יותר, 0-עדיפות הכי גבוהה. 0-99 אלו משימות real-time 100-140 אלו משימות אחרות מתועדפות פחות. כמו כן ישנו מנגנון שמנסה לעשות load balancing בין תהליכים עם אותה עדיפות. הוא מגדיר סט של מערכים, אחד לכל עדיפות – active array & expired array. תהליך שמקבל זמן מעבד אך נעצר באמצע לפני שניצל את כל הtime quantum שלו נשים אותו ב-active ואם הוא ניצל את כל הזמן שלו נשים אותו ב-expired. הבחירה תמיד תהיה מתוך הactive array, ברגע שהוא מתרוקן נהפוך בין התורים – נחזיר את כולם מתוך ה-expired לactive ונתחיל סבב חדש. רק כאשר 2 התורים יתרוקנו נוכל לתת שירות לתהליכים בעלי עדיפות גבוהה יותר – נמוכים יותר בתור.

ב JAVA יש לנו אפשרות להשתמש בפקודה שנקראת **yield** שמאפשרת לתהליך לוותר על זמן המעבד שלו לטובת תהליך אחר.

שאלות ממבחנים :

- שקופית 43 : עדיפות = מספר הכרטיסים שיש לתהליך.
  - א- אם נחלק לכל התהליכים את אותו מספר כרטיסים אז לכל התהליכים יהיה את אותו הסיכוי להיות מוגרלים בשלב הנוכחי ובממוצע נגיע לאותם הביצועים כמו ה-round robin. כל תהליך בממוצע יקבל 1/n מהמעבד.
  - ב- הבעיה שהוא פותר זו בעיית ההרעבה – גם אם העדיפות מאוד נמוכה עדיין יש את כרטיס ההגרלה שנותן לכל תהליך סיכוי לקבל זמן מעבד.
  - ג- אם התהליך רוצה לרוץ על מעבד ספציפי ולחכות עד לקבלת זמן מעבד ממנו הוא יעביר כרטיסים לתהליך אחר שאין לו בעיה לרוץ על המעבד הזה. כמו כן, כשנרצה שתהליך אחר יתקדם בצורה יותר מהירה ויבצע פעולה שהתהליך הנוכחי מחכה לה ולכן לא יכול לרוץ כרגע.
- שקופית 46 : נשים לב כי בשיטת ה-MLFQ, בכל פעם שתהליך מספר 1 יוצא ל I/O הוא חוזר לתור הראשון בהיררכיה וכן ברגע שהוא חוזר הוא מקבל ישר מענה.
- שקופית 50 : התשובות הן לסעיף ב'.

הרצאה 7:

סינכרוניזציה :

Race condition - שני תהליכים או יותר מנסים לגשת לאותו המשאב ולשנות אותו.
Critical section – קטע קוד הנוגע למשתנים המשותפים למספר תהליכים שעתידים להשתנות.
Entry section – בקשה לכניסה לקטע קוד קריטי.
Exit section – יציאה מהקטע הקוד הקריטי.
Remainder section – שאר הקוד שאינו הקוד הקריטי, שנמצא אחרי.

מנגנון שבו נשמש לפתירת הבעיה יכיל 3 פרמטרים :

- **Mutual exclusion** – מניעה הדדית, נרצה להיות בטוחים שרק תהליך אחד מריץ את הקטע בו זמנית.
- **Progress** – רק התהליכים שלא נמצאים בשארית הקוד, כלומר לא עברו את קטע הקוד הקריטי, יילקחו בחשבון בהחלטה של מי יריץ כרגע את קטע הקוד הקריטי. ההחלטה צריכה להתבצע כמה שיותר מהר.
- **Bounded waiting** - נרצה לייצר חסם על מספר התהליכים שיריצו את קטע הקוד הקריטי, מהרגע שתהליך מסוים הגיש בקשה להריץ את הקוד ועד שהוא נענה.

גישות להתמודדות עם קטע קוד קריטי במ"ה :

- **Nonpreemptive kernels** – תהליכים של ה kernel לא מקבלים זכות קדימה – כל פעם תהליך אחד רץ בkernel ולכן אין race condition.
- **Preemptive kernels** – ניתן להעדיף תהליך אחד על פני האחר בזמן הוא רץ בkernel, אך יש לדאוג שה kernelיודע להתמודד בצורה בטוחה עם race condition.

נעדיף את השיטה השנייה כי נרצה למנוע מצב שבו תהליך אחד רץ זמן ממושך על ה-kernel ומונע מתהליכים אחרים לרוץ. כמו כן מאפשר לתהליכים שצריכים לקבל תגובה ב-real time להיות מתועדפים.

- הפתרון של פיטרסון המיועד ל-2 תהליכים בלבד:

יש לנו 2 משתנים משותפים – turn ומערך של 2 דגלים. כשתהליך מגיע לקטע הקוד הקריטי הוא משנה את הדגל שלו במערך לtrue ואת המשתנה turn לתורו של התהליך הבא – j. ברגע שהתהליך יסיים להריץ את קטע הקוד הוא ישנה חזרה את הדגל שלו במערך לfalse.

הדגל במערך מסמן האם קטע הקוד הקריטי כרגע תפוס או לא, המערך מציין תור מי מבין התהליכים לגשת למשאב בפעם הבאה.

```
do {
  flag[i] = TRUE;
  turn = j;
  while ( flag[j] && turn == j);

  CRITICAL SECTION

  flag[i] = FALSE;

  REMAINDER SECTION

} while (TRUE);
```

- <u>פתרון מבוסס חומרה:</u>

נכריח את הפעולות שלנו להיות פעולות אטומיות וכך לא יקרה מצב שבו 2 תהליכים מריצים את אותו הדבר. ננעל את קטע הקוד, נריץ אותו ולאחר מכן נשחרר אותו.

יש 2 סוגים של פעולות אטומיות מבוססות חומרה(החומרה יודעת לתרגם אותן לפעולות אטומיות למרות שבפועל הן כוללות מספר שורות קוד – בוצע על ידי היצרן) :

- **TestAndSet** – אנחנו שולחים מצביע למשתנה בוליאני שאת ערכו נשנה. מובטח לנו שלאחר הרצה של פעולה זו, המשתנה הבוליאני יכיל true ויחזיר את ערכו <u>הקודם</u>.

```
boolean TestAndSet (boolean *target)
{
  boolean rv = *target;
  *target = TRUE;
  return rv:
}
```

```
do {
  while ( TestAndSet (&lock )); /* do nothing */
  // critical section
  lock = FALSE;
  // remainder section
} while ( TRUE);
```

- **Swap** – מחליפה בין 2 משתנים. נחזיק משתנה משותף שנקרא לו lock ונאתחל אותו לfalse. נחזיק משתנה לוקאלי בכל אחד מהתהליכים שנקרא לו key. נשתמש בלולאת while שמנסה כל הזמן להחליף בין הkey לlock. רק ברגע שנקבל חזרה false נדע שאנו יכולים להתקדם, כמו כן מובטח לנו שכל מי שניסה להריץ את הswap לפנינו יקבל חזרה true וידע שהוא צריך לעצור.

```
void Swap (boolean *a, boolean *b)
{
  boolean temp = *a;
  *a = *b;
  *b = temp:
}
```

```
do {
  key = TRUE;
  while (key == TRUE)
    Swap (&lock, &key );
  // critical section
  lock = FALSE;
  // remainder section
} while ( TRUE);
```

מנגנונים אלו עונים על כל התנאים מלבד התנאי של החסם – הם לא נותנים חסם למספר הפעמים שהתהליך יבקש את המנעול ויקבל אותו.

בשביל לפתור את בעיית החסם נוסיף מערך שנקרא waiting המשותף לכל התהליכים, כמו כן נוסיף משתנה lock – כולם מאותחלים ל-false. כשתהליך מסיים להשתמש במנעול הוא משחרר אותו ועובר על המערך על מנת למצוא תהליך אחר שמחכה למנעול. נעבור על המערך בצורה <u>מעגלית</u> עד שנחזור לתהליך הנוכחי. בצורה זו התהליך הבא בתור שמחכה למנעול יקבל אותו ונקבל חסם כגודל המערך שלנו.

לאחר ביצוע הקטע הקריטי התהליך יחפש במערך את התהליך הבא בתור שמחכה למנעול ע"י שינוי הסטטוס שלו במערך לfalse, וישנה את הlock לtrue. כלומר התהליך עצמו קבע מי הבא בתור שיריץ את קטע הקוד הקריטי.

החסם שלנו הוא n-1, מהרגע שתהליך ספציפי ביקש להיכנס הוא יחכה במקסימום ל- n-1 תהליכים אחרים שיריצו את הקטע לפניו.

```
do {
  waiting[i] = TRUE;
  key = TRUE;
  while (waiting[i] && key)
    key = TestAndSet(&lock);
  waiting[i] = FALSE;

  // critical section

  j = (i + 1) % n;
  while ( (j!=i) && !waiting[j] )
    j = (j + 1) % n;

  if (j == i)
    lock = FALSE;
  else
    waiting[j] = FALSE;

  // remainder section
} while (TRUE);
```

העיקרון הוא כזה – ברגע שהתהליך היחיד שמחכה הוא התהליך עצמו – זה שנמצא כרגע בתוך קטע הקוד הקריטי – הוא ישנה את המשתנה lock ל-false כי ממילא הנעילה תאפשר רק לתהליך אחד ויחיד להיכנס – לעצמו, ויעבור את לולאת הwhile באמצעות השינוי של key. ברגע שאנו יודעים שיש תהליכים אחרים שמחכים להיכנס לא נרצה לשחרר את הנעילה עצמה – כי אז כולם יוכלו להיכנס בבת אחת, לכן נשנה רק את המקום הרלוונטי במערך waiting להיות false מה שיגרום לתהליך הרלוונטי בלבד לעבור את לולאת הwhile ולהיכנס פנימה.

: Semaphores

```
Initialized to 1
do {

  wait(mutex);

  // critical section

  signal(mutex);

  // remainder section

}while (TRUE);
```

Wait() – מוריד 1 מהמנעול, **signal()** – מוסיף 1 למנעול.

Mutex = binary semaphore.

החיסרון – שהפקודה wait() ממושמשת באמצעות busy waiting/spinlock – שורף זמן מעבד.

נוכל לשנות את המימוש של wait() ו-signal() ע"י שימוש ב-2 תורים – waiting queue, ready queue, כאשר כל תהליך שרוצה לבצע wait() יכנס לתור של התהליכים שמחכים, וברגע שאחד המשאבים יתפנה, הוא יועבר ל-ready queue ע"י הפעולה signal().

```
typedef struct {
    int value;
    struct process *list;
} semaphore;
```

```
signal(semaphore *S) {
    S->value++;
    if (S->value <= 0) {
        remove a process P from S->list;
        wakeup(P);
    }
}
```

The magnit number of

The wakeup() opera execution of a block

```
wait(semaphore *S) {
    S->value--;
    if (S->value < 0) {
        add this process to S->list;
        block();
    }
}
```

The bloc process

Deadlock –תהליכים או יותר שמחכים בזמן בלתי מוגבל לevent שיכול להיווצר על ידי רק אחד מהם.

Starvation – נגרמת כתוצאה מכך שתהליך לא מקבל משאבים שהוא מחכה להם.

בעיות סינכרוניזציה מוכרות :

בעיות אלו יעזרו לנו לבדוק פתרונות אופציונאליים לבעיית הdeadlock.

- **בעיית הbounded-buffer** – נרצה לנהל buffer בעל n איברים שלתוכו אחד יכתוב והשני יקרא. נאתחל את הsemaphore ל-1, ניקח משתנים empty, full שגם הם מסוג semaphore על מנת שלא תהיה לנו בעיה בגישה אליהם ע ידי מספר משתנים. הערך של empty יהיה n כי בהתחלה יש לנו n מקומות ריקים. ערכו של full יהיה 0 בהתחלה.
התהליך שכותב יוריד את empty ב1 על מנת לבדוק שיש מקום לכתוב אליו את מה שהוא יצר. כמו כן הוא יוריד את הmutex ל1 ואז הוא יוכל לבצע כתיבה. כל תהליך אחר שיבוא בזמן זה אומנם יראה שיש מקום לכתוב אליו אך יחסם ע"י הmutex. ברגע ש-empty יגיע ל-0 ניתקע ולא נמשיך לmutex כי לא יהיה כבר לאן לכתוב. כאשר נסיים לכתוב לתוך המקום המסוים בבאפר נעשה signal לmutex – נשחרר את המנעול וכמו כן נעשה signal לfull על מנת לסמן שיש איבר נוסף שנקרא.
- **בעיית הכותבים-קוראים** – נניח שיש לנו DB שהרבה תהליכים מנסים לכתוב אליו. נרצה לוודא שבזמן שמישהו כותב האחרים לא קוראים ולא כותבים ובזמן שמישהו קורא מישהו אחר לא כותב בו זמנית. בכתיבה יש רק תהליך אחד ובקריאה יכולים להיות כמה כל זמן שאין כתיבה.
במשתנה readcount נספור כמה תהליכים כרגע בסטטוס של קריאה, זו האינדיקציה שלנו למתי נוכל לכתוב.
אם הreadcount ירד מ1 ל0 נשחרר את הנעילה בפני כתיבה. אם הוא גדל מ0 ל1 עלינו לנעול בפני כתיבה. נרצה לנעול גם את הreadcount שלא יבואו 2 תהליכים לשנות אותו במקביל.
כשתהליך ירצה לכתוב, עליו יהיה לתפוס את הmutex של write ופשוט לכתוב ובסוף לשחרר אותו.
כשתהליך ירצה לקרוא עליו קודם לחכות לmutex על מנת להגדיל את הreadcount ב1. הוא מתחרה על הmutex מול קוראים אחרים. ברגע שהוא קיבל גישה הוא מגדיל את readcount ב1. הוא שואל האם הוא היה 0 לפני שהגדיל אותו-> אם כן, עליו לדאוג שאין כרגע כותבים פעילים, איך יעשה זאת? -> הוא ינסה להשיג את הmutex של write. אם readcount גדול מ-1 אז אין לו צורך לדאוג לכלום. התהליך קורא עד שהוא מסיים ואז מנסה שוב לתפוס את הmutex על מנת להקטין חזרה את readcount. כעת עליו לדעת האם הוא הקורא האחרון על מנת לשחרר את הנעילה על הכתיבה.
- **בעיית ה"פילוסופים הסועדים"** – נגדיר מערך של 5 מנעולים. כל תהליך עושה wait ל-2 מנעולים אחד אחרי השני, מריץ את קטע הקוד ולאחר מכן משחרר את 2 המנעולים.
Deadlock – מספר תהליכים ינסו לנעול את אותו המנעול. פתרונות :
  - מערך של 4 מנעולים – כל תהליך יקבל 2 ואין בעיה.
  - לאפשר לתהליכים לתפוס מנעול רק אם המנעול השני פנוי גם הוא.

Deadlocks :

מצב של קיפאון\דריכה במקום – יש לנו 2 פעולות או יותר כך שכל אחת לא יכולה להתקדם מכיוון שהיא מחכה לאחרת. על מנת לפתור מצב זה עלינו לשחרר את המשאב.

starvation יכול להיגרם בעקבות deadlock אך זה לא מחייב, אלו שני דברים שונים ולכן מניעת השני לא בהכרח מונעת את הראשון. בdeadlock לא קיים scheduling שיוציא אותנו ממצב זה בעוד שבstarvation קיימת דרך יציאה אולם המערכת לא בוחרת בה לאורך זמן. ב-starvation תהליך כן מסוגל לגשת למשאבי המערכת בעוד שבdeadlock אף תהליך לא מסוגל לגשת למשאבים.

מודל מערכת – מורכבת מתהליכים ומשאבים לצורך ההמחשה, כאשר משאב יכול להיות :

- **Consumable** – משאב שניתן לצרוך פעם אחת.
- **Reusable** – משאב שניתן לצרוך יותר מפעם אחת.

4 תנאים חייבים להתקיים בשביל שיהיה לנו deadlock:

1. **Mutual exclusion** – רק תהליך אחד בו זמנית יכול להשתמש במשאב.
2. **wait&hold** – התהליכים מחזיקים לפחות משאב אחד וממתינים לקבל משאבים נוספים המוחזקים על ידי תהליכים נוספים. תהליך שאינו מחזיק משאב אינו חלק מהdeadlock.
3. **No preemption** - משאב ישוחרר מתהליך אך ורק על ידי התהליך עצמו לאחר שהוא סיים להשתמש בו.
4. **Circular wait** – המתנה מעגלית, קיימים לנו N תהליכים כך שכל אחד ממתין למשאבים המוחזקים על ידי התהליך הבא בתור, התהליך האחרון ממתין למשאבים של התהליך הראשון.

נתאר מצב נתון באמצעות גרף ובו יהיו התהליכים והמשאבים שלנו. חץ מתהליך למשאב מתאר מצב שבו התהליך מבקש גישה למשאב. חץ ממשאב לתהליך מתאר מצב שבו המשאב מוחזק על ידי התהליך.

אם בגרף אין מעגל -> אין Deadlock.

אם בגרף יש מעגל -> לא בהכרח יש deadlock. אם יש כמה מופעים של אותו המשאב המוחזק על ידי תהליך שנמצא במעגל, אז לא בטוח שיש לנו deadlock.

3 דרכים לטיפול בdeadlock :

1. **מניעה** – נבטיח שהמערכת לעולם לא תיכנס למצב של deadlock.
2. **התאוששות** – לא נמנע מהמצב הזה, נעלים עין אך אחת לכמה זמן נבדוק אם נוצר deadlock במערכת ואם כן נתאושש ממנו.
3. **התעלמות** – נעלים עין כל הזמן ולא נטפל. רוב מערכות ההפעלה משתמשות באופציה הזו.

- מניעה – תקיפת התנאים(גישה ראשונה) :

1. Mutual exclusion - נשתמש במשאבים כמו קובץ לקריאה בלבד, שאין לנו בעיה שמספר תהליכים ייגשו אליהם בו זמנית.
2. Hold&wait - ברגע שהתהליך יתחיל את ההרצה שלו הוא יבקש את כל המשאבים שהוא זקוק להם וכך יקבל את הכול בהתחלה ולא ימתין, הבעיה שזה מאוד בזבזני ויכול לגרום להרעבה.
3. No preemption - אם התהליך מבקש משאבים נוספים ממה שכבר יש לו, הוא בעצמו ישחרר את המשאבים שברשותו. תהליך יוכל לבקש משאבים ואם אין לנו אפשרות לתת אותם מידית הוא ישחרר את שלו כי כנראה תהליכים אחרים מחכים להם וברגע שיקבלו אותם הם ישחררו את המשאבים שהוא זקוק להם.
4. Circular wait - בקשת המשאבים תוכל להתבצע רק בסדר עולה, כלומר קודם נבקש את משאב 1, לאחר מכן את 2 וכן הלאה. זה פתרון מאוד בזבני אך מונע היווצרות של מעגל.

- מניעה – שמירת המערכת(גישה שנייה) :

אפשרות נוספת בתחום המניעה היא לבדוק בעת ההקצאה של משאב לתהליך כלשהו, אם המערכת עלולה להיכנס לdeadlock בעקבות הקצאה זו. המידע שנצטרך לדעת לשם כך הוא מה רמת המקסימום שכל משאב יצטרך מהמשאבים באופן רגעי – כמה משאבים לכל היותר הוא יצטרך ברגע נתון. נבנה אלגוריתם שישתמש ב-3 מדדים :

-מה מספר המשאבים הפנויים ברגע זה.

-מה מספר המשאבים שמוקצים כבר ברגע זה.

-מה יהיה מקסימום הדרישה של כל אחד מהתהליכים במהלך ריצתו.

נרצה לבדוק בכל שלב אם כתוצאה מההקצאה הנוספת שאנו הולכים לתת, קיים רצף של כל התהליכים במערכת, כך שלכל תהליך ברצף ניתן לספק לו את כל המשאבים שהוא צריך מתוך המשאבים הפנויים באותו הרגע ועוד את כל המשאבים המוחזקים על ידי התהליכים שלפניו ברצף. אם קיים רצף כזה->המערכת נמצאת במצב בטוח, אחרת, המערכת במצב לא בטוח.

הרצאה 8 :

אם המערכת במצב בטוח(safe state) -> אין Deadlock.

אם המערכת לא במצב בטוח(unsafe state) -> ייתכן ויש deadlock.

אם יש רק מופע אחד מכל resource type אז נשתמש בגרף, אחרת-אם יש יותר ממופע אחד של כל resource type אזי נשתמש באלגוריתם של בנקר.

האלגוריתם של בנקר :

נניח שיש n תהליכים ו- m סוגי משאבים.

הנתונים :

- מערך available שבו בכל תא מספר המופעים שיש לנו מכל סוג משאב.
- מטריצה max n*m ובה עבור כל תהליך, מה הדרישה המקסימלית שלו לכל משאב.
- מטריצה allocation n*m ובה עבור כל תהליך, כמה מופעים מוקצים לו ממשאב מסוים.
- מטריצה need n*m ובה עבור כל תהליך כמה מופעים נוספים ממשאב מסוים הוא צריך על מנת לסיים את המשימה שלו.
  Need = max –allocation.

נרצה להעריך האם המערכת נמצאת במצב safe state.

שלב 1 - נאתחל מערך work בגודל סוגי המשאבים - m שברשותנו, נשים בו את available – כמות המשאבים שפנויים כרגע מכל סוג. כמו כן נאתחל מערך finish בגודל כמות התהליכים שיש לנו – n ונשים בו false.

שלב 2 – נמצא תהליך כלשהו כך שלא סיימנו לטפל בו עד עכשיו וגם כמות המשאבים שהוא צריך קטנה שווה מכמות המשאבים הפנויים שיש לנו כרגע.

שלב 3 – נעדכן את מאגר המשאבים הפנויים שיש לנו ונעדכן שהתהליך סיים.

נמשיך שוב ושוב עד שלא נמצא בשלב 2 תהליך מתאים. ואז נגיע ל:

שלב 4 – נשאל האם כל התהליכים סיימו. אם כל המערכת נמצאת במצב Safe אחרת המערכת נמצאת במצב unsafe.

האלגוריתם יחפש בכל פעם את התהליך הבא שעוד לא סיים טיפול וגם שיש ברשותנו מספיק משאבים עבור ריצתו. בהנחה שמצאנו כזה, לאחר סיום ריצתו משאביו ישוחררו חזרה למערך work ונבדוק האם קיים כעת תהליך חדש שעבור המשאבים הפנויים ועבור המשאבים שהוחזקו על ידי התהליכים שלפניו כעת הוא יוכל לרוץ.

<u>דוגמת ריצה:</u>

בהתחלה נאתחל את work לפי הavailable – (A, B, C מהווים סוגי המשאבים שיש לנו)

- A = 3
- B = 3
- C = 2

ונאתחל את Finish למערך של חמישה תהליכים, שכולם False. כעת נחפש איזה תהליך ניתן לספק. ניתן לספק את P1 ואת P3. כי הNeed שלהם קטן מהWork. (את הNeed בנינו מההפרש של Max פחות Allocation). נגיד ניקח את P3: נפנה לערך שלו בAllocation (2,1,1) ונוסיף אותו לWork. קיבלנו 5,4,3. כעת סיפקנו את P3, נשנה את התא המתאים בFinish לTrue. כעת באותו אופן נמשיך עם שאר התהליכים ונספק את כל התהליכים. סדר הסיפוק הוא – P3 -> P4 -> P1 -> P0 -> P2.

מכיוון שהגענו בסוף בWork לכמות המשאבים הכוללת, וגם כל התהליכים סופקו – הגענו ל Safe State לפי האלגוריתם של בנקר.

האלגוריתם הבא בודק האם אנחנו יכולים לענות לבקשה של תהליך שמבקש משאבים נוספים. כל פעם שתהליך יבקש עוד משאבים נריץ את האלגוריתם הנ"ל כדי לבדוק האם ההיענות לבקשה תשאיר אותנו במצב safe.

אם אחרי ההרצה נהיה בsafe state – נוכל לתת לו את המשאבים. אם לא – הוא ייאלץ להמתין.

בהתחלה במערך Request תהיה הבקשה של התהליך, ובמקום הj יהיה הK שהוא מספר הintstanceים של משאבים שהוא מבקש.

1. אם מס' המשאבים שהוא ביקש, קטן או שווה למס' המשאבים שהוא צריך – נלך לשלב 2. אחרת התהליך שיקר במקור ולכן נדפיס שגיאה.
2. אם מספר המשאבים שהתהליך מבקש קטן שווה ממספר המשאבים הפנויים – נלך לשלב 3, אחרת התהליך יאלץ לחכות.
3. נעדכן "בכאילו" את המערכים בהתאם למשאבים שהקצנו בסיבוב זה ונריץ את האלגוריתם הקודם שיבדוק האם אנחנו עדיין בsafe אם כן – המשאבים יוקצו, אחרת התהליך יאלץ לחכות.

<u>שיטת ההתאוששות :</u>

נצטרך שתהיה לנו דרך לזהות deadlock – את הבעיה עצמה, כי אנחנו מאפשרים למערכת להיכנס לבעיה זו.

הדרך שלנו לזהות deadlock במידה ויש לנו משאב אחד מכל סוג היא באמצעות גרף. כל קדקוד מייצג תהליך. חץ בין Pi ל Pj אומר ש Pi מחכה לתהליך Pj. עכשיו נוכל לנסות ולמצוא בגרף מעגלים כי

אם יש לנו מעגל במצב שבו יש משאב אחד מכל סוג אזי יש deadlock. בהתאם נכריז אם יש או אין בעיה.

כאשר יש לנו מספר משאבים מכל סוג נשתמש ב-3 מערכים : work, finish, request ואלגוריתם.

אם יש תהליך שלא מחזיק שום משאב אזי הוא לא חלק מהdeadlock ונסמן במקום המתאים במערך true – finish.

ניהול הזיכרון - Main Memory

זיכרון פיזי: איפה שבאמת נשמרים הנתונים שלנו (RAM), לכל נתון יש כתובת בזיכרון. מחולק לwords ובו מאוחסן הdata וה-instructions שלנו.

זיכרון לוגי: (או ווירטואלי) הוא טכניקה לניהול והקצאה של זיכרון המחשב ,המסתירה את הזיכרון הפיזי של המחשב ומדמה זיכרון רציף וגדול, ומפרידה בין ניהול הזיכרון של תהליכים שונים. כל אחת מהתוכניות המתבצעות פועלת כאילו עומד לרשותה מרחב זיכרון בגודל שהיא זקוקה לו, רציף, וללא הפרעות מתהליכים (לא מוזמנים) אחרים. הוא יעבוד עם כתובות וירטואליות בטווח 0 עד מרחב הכתובות של התהליך.

הבדל גדול ביניהם הוא שייתכן שפיזית התהליך יפוזר בזיכרון הפיזי, אך מבחינת הווירטואלי אנו מסתכלים עליו במרחב זיכרון רציף (כלומר מוקצים תאים רציפים בו).

בזיכרון הווירטואלי המיקום עליו אנו מסתכלים הוא תמיד ביחס לנקודת ה-0 ממנה מתחיל מרחב הזיכרון, לעומת זאת בזיכרון הפיזי הכתובת היא תמיד יחסית בהתאם לכתובת ההתחלתית שמערכת ההפעלה הקצתה לנו. הזיכרון הווירטואלי נועד ליצור מרחב כתובות רציף מנקודת 0 לצורך נוחות התהליך.

מרחב הכתובות הווירטואלי לא באמת קיים! הוא לא נשמר בשום מקום! הוא קיים רק בלוגיקה של ריצת התהליך.

באיזה חלק נעשה ה Binding (צימוד/מיפוי) בין הinstructions וה- data לזיכרון? באחד משלושת החלקים הבאים:

1. **בזמן הקימפול** – נקמפל את הקוד, נקבל תכנית שבתוכה כבר רשומות הכתובות הפיזיות בזיכרון הראשי. למה זה טוב? זה חוסך את כל התרגום מווירטואלי לפיזי. זה יותר בטוח. אף תכנית לא תוכל לגעת בזיכרון של תכנית אחרת. זה יקרה אם נדע מראש איפה התכנית תישמר. נשמור את הקוד עם כתובות אבסולוטיות, וזה נקרא "קוד אבסולוטי". אם נרצה להריץ את הקוד במקום אחר בזיכרון – נצטרך לקמפל מחדש. במקרה הזה הזיכרון הפיזי והווירטואלי חופפים.
2. **בזמן ה-Loading** - (בזמן הטעינה והיצירה של התהליך בתכנית) ניקח טווח כתובות פנוי בזיכרון הפיזי, וכך נייצר ממנו את הכתובות הווירטואליות. למה זה טוב? הרווחנו גמישות, וניצולת יותר טובה של המשאב, כיוון שהתכנית תוכל לרוץ איפה שנבחר, אך לא תוכל לזוז בזמן ההרצה. אין צורך לעשות תרגום מווירטואלי לפיזי, אך מצד שני לא ניתן להזיז אותה.
3. **בזמן הExecution** - (ההרצה) התרגום יתבצע בRun Time , וזה מאפשר "להזיז" את התכנית למקום אחר בזיכרון בזמן ההרצה.

אנו מבינים שהכתובת הלוגית והפיזית הן <u>זהות</u> בשני השלבים הראשונים, והן <u>שונות</u> בשלב השלישי.

MMU - device שמבצע את המיפוי בין כתובת ווירטואלית לכתובת פיזית. בסכמה המבוססת MMU יהיה **Relocation register** – קבוע שעוזר לנו לבצע את ההמרה (שיגיד לנו מאיפה מתחילה התכנית שלנו בזיכרון הפיזי בהנחה שהזיכרון רציף) והוא יתווסף ל**Logical Address** (דוגמה 346) הRelocation register (דוגמה 14000), ונקבל את הכתובת הפיזית (דוגמה 14346). התכנית של המשתמש תתעסק רק בכתובות לוגיות, ולא פיזיות.

Swapping – לקיחת תהליך מהזיכרון (שיכול להיות במצב ready) והעברתו ל**Backing store** (מתקן אחסון – למשל הארד דיסק שמספיק גדול בשביל שנוכל לשמור בו את התהליך ולשלוף כשנצטרך). הSwapping קורה במקרים שאין לנו מספיק זיכרון לתהליכים. נעשה Swap out לאחד מהתהליכים ונעביר לBacking store. איך יישמר בדיסק? כImage, מעין תמונת זיכרון. בדר"כ נשתמש כשיש תהליכים עם Priority Scheduling, ואז כשמגיע תהליך עם עדיפות גבוהה- נעשה **swap out** ו **swap in**. חלק גדול מהזמן שלוקח לעשות Swapping זה בהעברות עצמן, כיוון שמהירות הדיסק יותר איטית ממהירות הזיכרון.

אנו נחזיק ready queue של כל התהליכים שמאוחסנים בזיכרון ומוכנים לרוץ.

Contiguous Allocation :

הזיכרון הראשי מתמפה ל-2 סוגים :

- **Low memory** – החלק בו מאוחסנים תהליכים של מערכת ההפעלה.
- **High memory** – החלק שבו מאוחסנים תהליכי המשתמש.

לצורך כך נחזיק 2 רגיסטרים :

- **Base register** – מחזיק את הערך של הכתובת הראשונה בזיכרון הפיזי בהנחה שהוא רציף (מקביל ל-relocation register).
- **Limit register** - מחזיק את הכתובת הגבוהה ביותר בזיכרון הווירטואלי.

בהינתן כתובת וירטואלית, נבדוק שהיא קטנה מהכתובת הגבוהה ביותר – limit register, ובמידה וכן נוסיף לה את הכתובת ההתחלתית בזיכרון הפיזי ונגיע לכתובת הפיזית הרלוונטית. במידה ולא, נייצר Trap עבור שגיאה.

שמירת התהליכים בצורה רציפה בMain Memory. ברגע שתהליך מסיים את פעולתו הזיכרון שהוא תפס מתפנה -> יש לנו חור בזיכרון ובמקום זה נוכל לשבץ תהליך אחר אבל עדיין יישאר לנו חור קטן. מה שיקרה זה שברגע שתהליך מפנה את מקומו בזיכרון, במקומו ייכנס תהליך אחר השווה או קטן לו בגודלו ולכן ברב הפעמים יישארו לנו חורים קטנים שלא נוכל להשתמש בהם. אמנם באיחוד החורים יחד נוכל לקבל מספיק מקום לאחסון תהליך חדש, אך בשיטה זו אנו משתמשים בזיכרון רציף.

איך נקבל את ההחלטה על איזה תהליך לשבץ באיזה חור? יש 3 שיטות :

- **First fit** – מבין החורים הקיימים, ניקח את החור הראשון שמספיק גדול להכיל את התהליך שלנו. שיטה זו מהירה מאחר והיא לא מצריכה אותנו לעבור על כל החורים הקיימים.
- **Best fit** – עובר על כל החורים בזיכרון ולוקח את החור שהוא הקטן ביותר שעדיין מסוגל להכיל את התהליך שלנו.
- **Worst fit** – נעבור על כל החורים ונבחר את החור הגדול ביותר. למה שנעשה זאת? – ניקח בחשבון שהתהליך יכול לגדול, כמו כן זה ישאיר לנו חור עדיין גדול מספיק לאחסון של תהליך נוסף. בדרך כלל שיטה זו פחות טובה כי היא מסתמכת על כך שהחור יספיק ליותר מהתהליך אחד.

בעיות :

פרגמנטציה חיצונית – מצב בו יש לי מספיק זיכרון בין תהליכים פנוי לאחסון תהליך אך הוא מפוזר על פני מקומות שונים בזיכרון ואינו רציף ולכן לא ניתן להשתמש בשטח זה לאחסון התהליך.

נתמודד באמצעות **compaction** – ניקח את כל התהליכים ונתחיל לדחוף אותם כלפי מעלה בצורה כזו שהם ישבו בצורה רציפה אחד אחרי השני וכך נסגור את כל החורים. זה תהליך שלוקח המון זמן כי זה אומר לקרוא את כל תאי הזיכרון ולכתוב אותם מחדש, כמו כן לפעמים אנו נותנים להתקן חיצוני מקום בזיכרון והוא עובד מולו בצורה עצמאית. אם נעשה compaction זה יגרור מצב שבו ההתקן החיצוני אינו יסונכרן אל מול המקום החדש שלו בזיכרון. נוכל למנוע ע"י שימוש ב-באפרים של מערכת ההפעלה או ביצוע פעולת הcompaction תוך בדיקה שאין פעולות i/o שמתבצעות תוך כדי.

פרגמנטציה פנימית – מצב בו נותנים לתהליך זיכרון יותר ממה שהוא בפועל צריך להשתמש. בתוך כל המקום שהוקצה לתהליך – יש מקום לא מנוצל.

Paging – דרך נוספת לפתור את בעיית הפרגמנטציה החיצונית בשיטה הרציפה. נקבע יחידה בסיסית בגודל קבוע – **page** (לדוגמא 2^9 בתים). את הזיכרון הפיזי שלנו נחלק ל**frames** ששקולים בגודלם לגודל הpageים. כמו כן, את הזיכרון הווירטואלי נחלק גם כן לpageים.

כאשר נרצה לפנות לכתובת כלשהי בזיכרון הווירטואלי – נבדוק מה הpage שבטווח הכתובות שלו נמצאת הכתובת המבוקשת וניקח את ה**offset** מכתובת ההתחלה של הpage עד לכתובת המבוקשת. איך תראה הכתובת שתחזור? – P ביטים המייצגים את האינדקס של הpage בתוכו נמצאת הכתובת + d ביטים המייצגים את הoffset בתוך הpage.

כעת, אנחנו לא באמת צריכים לשמור את הframeים זה אחר זה בצורה רציפה בזיכרון אלא חשוב לנו שרק הframe עצמו ישמר בצורה רציפה מאחר וגודלו כגודל הpage. כל מה שאנו צריכים זו טבלת המרה פשוטה – **page table** , שתאמר לנו עבור כל Frame לאיזה Page בזיכרון הוא משויך. הטבלה הזו תחזיק לנו את המיפוי. לכל תהליך יהיה page table משלו.

בהינתן page ו-offset, נחליף page ב-frame ע"י טבלת ההמרה הרציפה השמורה בזיכרון, שם באינדקס ה-p שמור המקום של הframe בזיכרון הפיזי אליו נוסיף את הoffset על מנת להגיע למקום הרצוי בזיכרון.

תהליך שמסיים את פעולתו מפנה את הframeים שהוא תפס. ה"חור" שהתפנה ייתן לנו מכפלה כלשהי של frameים שיכולה לשמש אותנו לאחסון של תהליך אחד לפחות. לכן, מצד אחד פתרנו את בעיית הפרגמנטציה החיצונית שהיא הקשה יותר – אין לנו יותר חורים שלא נוכל להשתמש בהם! אבל, יש לנו עדיין פרגמנטציה פנימית אך זה קורה רק כאשר גודל התהליך קטן משמעותית מגודל הpage.

מערכת ההפעלה צריכה לשמור את הframeים הפנויים שאינם מוקצים וניתן להשתמש בהם. ברגע שיש תהליך חדש, ניגש לרשימה הframeים הפנויים ומשם נקצה את הframeים הראשונים הדרושים לריצת התהליך החדש.

על מנת לנהל את הpage table בזיכרון עלינו להשתמש ב2 רגיסטרים:

- **PTBR – page table base register** – שומר את כתובת הpage table.
- **PRLR – page table length register** – שומר את גודל הטבלה.

בעיה - בגלל שהטבלה נשמרת בזיכרון, כל גישה שלנו לזיכרון(ל-block בזיכרון הפיזי) מחייבת גישה נוספת לזיכרון לטבלה עצמה) - על מנת להבין לאיפה בזיכרון הפיזי עלינו לגשת. הגדלנו את זמן

הביצוע פי 2- בעיית מהירות. בכל גישה לזיכרון אנו בעצם מייצרים 2 גישות לזיכרון. הכפלנו את זמן ההרצה של התוכנית כמעט פי 2! (רק בהקשר של הגישה לזיכרון).

על מנת לפתור את בעיית המהירות נייצר cache לטבלה (בנוסף לטבלה הקיימת) – **associative memory** או בקיצור **TLB**.

עלינו להשתמש בטבלה עם 2 עמודות לpages ול-frames. בהינתן מספר page כלשהו, נשתמש בטבלה הנתונה על מנת לקבל את המיפוי ל-frame המתאים. אם המיפוי נמצא בטבלה – **hit** – פשוט ניגש אליו, אחרת – **miss** – ניגש לpage table על מנת לשלוף ממנה את מיקום הFrame המתאים.

הגנה על הזיכרון תתבצע ע"י הוספת ביט לpage table. הביט הזה נקרא **valid-invalid bit**. הביט הזה יסמן האם הpage הספציפי valid – מוקצה עבור התהליך, או שהוא לא באמת קיימת במרחב הכתובות של התהליך שלנו.

Shared code – נוכל לחלוק חלקים מהזיכרון הפיזי של התהליך שלנו עם תהליכים אחרים בצורה מאוד נוחה במקום לשכפל את אותו המידע לחלקים שונים בזיכרון.

לא היינו רוצים לשמור את הpage table בצורה רציפה בזיכרון כי זה נוגד את כל העיקרון שאנו עובדים לפיו. ישנם 3 פתרונות שאמורים למנוע את בעיית הופעת page table גדולה ורציפה בזיכרון :

- **Paging hierarchy** – נפרק את ה page table למספר טבלאות שלא ישבו בצורה רציפה זו אחר זו בזיכרון. נשתמש בטבלה חיצונית שתשלוט על הטבלאות הקטנות כלומר, במקום הראשון בטבלה הזו תהיה לנו הכתובת של תת הטבלה הראשונה של הpage table וכן הלאה.
  כעת, נוצר מצב שכל גישה לזיכרון היא במחיר של 3 גישות ובעיה זו תגדל כל שנגדיל את ההיררכיות שלנו.
- **Hashed page table** – נשתמש בפונקציית Hash – למשל מודולו 10. בעזרתה נגיע למקום המתאים בpage table שם נמצא את כתובת הframe המתאים. מכיוון שלא נשתמש באינדקסים על מנת להגיע למקום המתאים בטבלה, אלא נשתמש בפונקציה שתמפה אותנו למקום המתאים, אין לנו צורך לשמור את הטבלה באופן רציף בזיכרון. בכל פעם שפונקציית ה-hash לא תגיע למקום הנכון (כי יש לנו כמה מספרים שממופים לאותו הדבר) זו תהיה נגיעה בזיכרון.
- **Invert page table** – נשמור את כל הטבלאות של כל התהליכים בטבלה אחת גדולה, כאשר נצמיד לכל חיפוש שנבצע ערך נוסף – pid של התהליך. אומנם הטבלה עדיין תישמר ברציפות בזיכרון אך זו טבלה אחת המשותפת לכל התהליכים. כמו כן זה דורש מאתנו לעבור על כל הטבלה על מנת למצוא את הpid המתאים עם ההמרה המתאימה.

אלטרנטיבה לניהול הזיכרון :

סגמנטציה – באמצעות הpageים ביצענו חלוקה שרירותית של מרחב הזיכרון שלנו. נרצה שיהיה לנו סדר לוגי בזיכרון ע"י חלוקה ליחידות לוגיות כרצוננו.

חלוקה של התוכנית ל"מקבצים לוגים", כלומר main נכנס לsegment אחד, משתנים גלובליים נמצאים בסגמנט אחר. סגמנטים מנוהלים באותו אופן שבו מנוהלים pages (base, limit).

Base – הכתובת ההתחלתית של הsegment בזיכרון הפיזי.

Limit – גודל הsegment הספציפי.

הקריאה מכל מקבץ, תהיה עם Offset מסוים. ה segment נשמר בצורה רציפה, הבעיה שיכולה לצוץ היא External fragmentation.

נשתמש בPageים- נקצה זיכרון ל-segment בכפולות של Page כך שיכיל את כל הsegment. הsegment יפוזר בתוך frameים בזיכרון, ולא מעניין אותנו איפה. נשתמש בטבלה כדי לדעת איפה החלקים שלו שמורים. כל segment למעשה הוא תהליך שממומש לפי Pageים. איך יעבוד המימוש? תהיה לנו טבלת Pageים לכל segment (כי אנו מסתכלים על כל הsegment בזיכרון כרציף), למשל נצטרך תא זיכרון של s segment בd offset, ובטבלה נמצא את הכתובת של הPage table שמתאים לאותו s segment, איזה Page נצטרך? לא ידוע. לכן ניקח את d (הכתובת מהטבלה) ונחלק אותו בגודל הsegment (שזה גודל הpage) – השלם נותן את מס' הpage, והשארית נותנת את הOffset בתוך הpage. הOffset זה הd' וה p זה הPage. וכך נחשב את הכתובת בMain Memory אותה חיפשנו.

סך הכול יש לנו שלוש נגיעות בזיכרון – הSegment table, הPage table ואז בMain Memory.

הרצאה 9 :

Virtual Memory

הרעיון הוא הפרדת הLogical Memory מהPhysical Memory הפרדה מוחלטת. אם נריץ תכנית ונייצר ממנה תהליך, התהליך הזה יהיה בעל מרחב כתובות וירטואלי (לוגי) משלו, ומולו צריכה לשבת כתובת בזיכרון הפיזי. מהיום נשתמש בVirtual Memory כך שרק **חלק** ממה שמיוצג ע"י מרחב הכתובות הלוגי – באמת יישב בזיכרון הפיזי. השאר יישב באיזה שהוא storage שבעת הצורך נביא אותו משם. במלים אחרות – אין צורך שכל התהליך יישב כל הזמן בזיכרון, אלא רק החלקים הרלוונטיים. מדוע זה טוב?

1. אם נשים בזיכרון רק חלק מהתהליך, נוכל להשתמש באותה כמות של זיכרון בשביל להריץ יותר תהליכים בו זמנית. למשל אם יש מחשב שמשמש משתמש אחד – המשתמש יוכל לקנות מראש פחות זיכרון.
2. במקרה שנרצה להתקין משהו שצורך יותר זיכרון ממה שיש לנו, נוכל להריץ תוכנות שסך הזיכרון שהן צריכות גדול מסך הזיכרון הפיזי שמותקן במחשב שלנו.
3. יצירת התהליך – תהליך כבד, אם לא נעבוד עם ה Virtual Memory, נצטרך לחכות לLoader שיעלה את כל החלקים של התוכנה. אם נשתמש בשיטה זו, הוא יעלה רק את החלקים הרלוונטיים. נפסיד אמנם בזמן הריצה, כי נגלה שנצטרך להעלות עוד דברים, ואז זה קצת יעכב אותנו.
4. צריכת פחות I/O - במקרה שיש זיכרון מאוד גדול, ונביא בכל פעם שנצטרך, נצרוך פחות I/O. בעייתי במקרה שיש מעט זיכרון, ואז נביא בכל פעם אך גם נצטרך לפנות. ובכך השיטה לא תהיה יעילה.
5. צריכת פחות זיכרון לתהליך.
6. הResponse time יותר מהיר – רק בזמן הפעלת התהליך, במהלך הריצה יהיה יותר איטי.
7. מתן שירות ליותר משתמשים.

מימוש: דרך Demand paging או Demand segmentation. (אנו נלמד את Paging, אך המעבר מידי)

Demand Paging:

נביא page לזיכרון, ונאחסן באחד מframeים ונשלוף אך ורק כשנצטרך אותו. נעשה זאת בעזרת **Swapper**, (אנו נשתמש בLazy Swapper)– שולף רק מה שצריך כשצריך. Swapper שעובד עם pageים נקרא Pager.

אם נצטרך page – נעשה רפרנס לאותו page. אם הרפרנס חוקי (=page באמת במרחב הכתובות הלוגי) אז נבדוק האם הוא נמצא בזיכרון. אם כן – ניגש אליו בצורה ישירה. אם לא – נצטרך להביא אותו לזיכרון (ייתכן ונצטרך להוציא מישהו מהזיכרון לשם כך).

בתרשים מוצגת פעולת הSwapping: ניקח תהליך, נשים אותו בדיסק, הוא כבר לא יהיה ב Ready Queue. ואותו דבר להיפך, נוכל לשלוף מהדיסק תהליך לזיכרון, ואז הוא יוחזר לReady queue אם היה מלכתחילה שם.

לטבלת הpage נצטרך להוסיף ביט של Valid או Invalid, שייצג האם הpage הזה נמצא כרגע בזיכרון או שצריך להביא אותו מהדיסק. דוגמה בשקף שלאחר מכן – הPage table נותן לנו את התרגום בין ה Logical Memory לבין הPhysical Memory. בדיסק אנו רואים את כל הpage ים שמורים גם שם. מה ההבדל בין פייג' 0 (A) לפייג' 1 (B) – אחד Valid והשני Invalid, כלומר ש A לא מעודכן ו B כן.

מדוע אנו שומרים לכולם בלוקים בדיסק בצורה רציפה ולא רק לאלה שהם Invalid? כיוון שנרצה להשתמש בזה בצורה הכי קרובה לRandom Access, כלומר אם עכשיו נצטרך לאחסן את C כדי לפנות אותו מהזיכרון, לא נרצה לחפש מקום או להבין איפה הוא נמצא, אלא להגיע כמה שיותר מהר לדיסק ולרשום אותו.

מה קורה כשמבקשים את הרפרנס הראשון לפקודה הראשונה של התהליך? יהיה **Page Fault**. למה? כי שום דבר עדיין לא נמצא בזיכרון. הטבלה כולה Invalid בהתחלה. או מה יקרה אם ניסינו להגיע לPage ובטבלה מופיע Invalid יהיה **Page Fault** . מ"ה תצטרך להבין מדוע זה קרה, היא תבדוק בטבלה צדדית האם מדובר בpage לגיטימי, במידה וה-page לגיטימי וחוקי – אז בעצם הוא לא נמצא בזיכרון. נצטרך למצוא בזיכרון פריים פנוי (Free frame), ואז נעשה Swap לpage מתוך הדיסק לתוך הFree Frame הזה. ואז נלך לרשומה המתאימה עם הInvalid בPage Table, נשנה את הInvalid לValid, ונעשה אתחול מחדש ל instruction שעשתה את הPage Fault.

איך זה משפיע על הביצועים שלנו? (Effective Access Time – EAT תוחלת הזמן שלוקח לנו בשביל להגיע לכתובת בזיכרון) כדי לחשב זאת – ניקח משתנה p שייצג את האחוז הפעמים שביקשנו פייג' והוא לא היה בזיכרון (Page Fault rate). כלומר אם P=0 – אז אף פעם אין page fault (כל התהליך נמצא בזיכרון למשל), ואם P=1 אז עבור כל בקשה נגיע ל Page Fault.

אזי הEAT יהיה לפי החישוב שבשקופית. (1-p) זה כל המקרים שלא היה Page Fault, נכפיל את היחס הזה בזמן גישה לזיכרון, ונחבר את ההסתברות שייקח לנו לטפל לPage Fault (Overhead) של ההבנה שקרה Page Fault, ועוד הזמן שייקח לנו להביא את הpage מהדיסק לפריים הפנוי בזיכרון, ועוד פוטנציאלית הזמן שייקח לנו למצוא פריים שלא פנוי בזיכרון ולהוציא אותו מהזיכרון ולהחזיר לדיסק, ונוסיף את הזמן של לעשות אתחול לפקודה).

<u>דוגמה מהמצגת</u>: נניח שזמן הגישה לזיכרון הוא 200 ננו שניות, נניח שכל הזמן ה EAT ייקח 8 מילי שניות. נחשב כעת את ה EAT כמו בשקופית (טכני). הפרמטר בסוף שישפיע הוא p. אם p למשל יהיה 1/1000 (אינטואיטיבית זה p ממש טוב), נציב ונקבל EAT של 8.2 מיקרו שניות (8200 ננו שניות), אם לא היינו משתמשים בזיכרון - הEAT היה גדול כמעט פי 40 מזמן הגישה לזיכרון בלי הVirtual Memory.

<u>מה עוד טוב בVirtual Memory?</u>

1. **Copy-on-Write**: כשיש תהליך-אב למשל שמייצר תהליך-בן (fork), אנו משכפלים את מרחב הזיכרון של האב בשביל הבן, אך כשעובדים בVirtual Memory לא חייבים לעשות

זאת. נוכל לקחת את התהליך אב, ולמפות את הבן לאותם pageים בדיוק. כעת שניהם יוכלו להתקדם מבחינת ההרצה עם אותו תוכן של pageים שנמצאים בזיכרון, עד שאחד מהם יצטרך לשנות את אחד מהPageים. ברגע שאחד ירצה לעשות write – נשכפל את הpage ונקשר את התהליך שרצה לכתוב לפייג' החדש. כך נחסוך מקום בזיכרון וזמן שכפול של התהליך.

במידה והיה Page fault, ונצטרך לחפש free frame ולא מצאנו אחד כזה, נצטרך להוציא מישהו החוצה, איך נעשה זאת? נצטרך אלגוריתם שיגיד לנו את מי כדאי להוציא. נניח ויש כמה אלגוריתמים שניתן לבחור מתוכם, איך נעריך איזה הכי טוב? מה נרצה למקסם או למזער? נרצה להגיע למינימום Page faults.

איך יעבדו האלגוריתמים? נמצא את הpage שנרצה להביא מהדיסק, נמצא frame פנוי, אם יש כזה נשתמש בו, אם לא- נקרא לאלגוריתם והוא יבחר **Victim** (קורבן) שיצטרך להתפנות, נוציא את הקורבן מהזיכרון לדיסק ונעדכן בטבלה שהוא Invalid, נביא את התוכן של הpage שאליו נרצה לגשת אל הפריים שהתפנה, נעדכן את הPage table עבור שני הpage ים האלה, ונעשה restart לתהליך. כל זה בתרשים Page Replacement.

<u>יש מספר אלגוריתמים :</u>

- **FIFO– First in first out** – בשיטה זו הקורבן יהיה ה page הראשון שהועלה לזיכרון. השיטה הזו לא עומדת בציפייה שלפיה יותר frameים פנויים -> פחות page faultים.
- **האלגוריתם האופטימלי** – מבטיח מינימום כמות של page fault. לפיו הקורבן יהיה הpage שנשתמש בו עוד הכי הרבה זמן – נסתכל על כל הPageים שכבר נמצאים בזיכרון ונבחר את הpage שלפי המחרוזת אנו עתידים לבקשה בנקודת הזמן הכי רחוקה מנקודת הזמן הנוכחית ואותו נחליף. שיטה זו לא ישימה – לא נוכל לחזות את העתיד.
- **LRU – least recently used** – הקורבן יהיה הpage שלא השתמשנו בו הכי הרבה זמן.
<u>2 שיטות למימוש :</u>
- נשמור חותמת זמן לכל page.
- נשתמש במחסנית = רשימה מקושרת דו כיונית. ברגע שיש פנייה לpage מכניסים אותו לראש הרשימה. הקורבן יהיה הpage האחרון ברשימה.
- **LRU Approximation –** נתחזק מערך של ביטים שמאותחל לאפסים. עבור כל גישה ל page ניגש למקום המתאים במערך ונשנה את הביט ל-1. הקורבן יהיה אחד מהpageים שהביט שלו בזמן זה עדיין מאופס. נאפס את המערך אחת לכמה זמן.
- **Second chance -** נתחזק מערך של ביטים שמאותחל לאפסים. עבור כל גישה ל page ניגש למקום המתאים במערך ונשנה את הביט ל-1. נתחזק מצביע למקום שרירותי במערך זה. בכל פעם שנחפש קורבן, תחילה נבדוק על מי המצביע עומד. אם הוא עומד על תא שיש בו 0 הוא יהיה הקורבן שלנו ואותו נחליף. אחרת, הוא עומד על תא שיש בו 1 – את ה-1 נשנה ל-0 (ניתן לו הזדמנות נוספת) ונתחיל לנוע על המערך בכיוון השעון במטרה למצוא תא מאופס כאשר בדרך נהפוך כל ביט שהוא 1 ל-0. במידה ומצאנו תא מאופס ניקח אותו לקורבן, אחרת נחזור לתא הראשון שאיפסנו ונבחר בו.

<u>**Counting algorithms -**</u> באלגוריתמים אלו נשמור מונה לכל page עבור מספר הגישות שביצענו אליו עד כה.

- **LFU –** הקורבן יהיה בעל המונה הכי <u>נמוך</u> – זה שפנינו אליו הכי מעט -> כנראה שהכי פחות נצטרך אותו.
- **MFU –** הקורבן יהיה בעל המונה הכי <u>גבוה</u> – ההנחה היא הpageים שפנינו אליהם הכי מעט, נצטרך לגשת אליהם שוב.

הרצאה 10 :

Equal allocation – כל תהליך יקבל מספר שווה של frameים.

Proportional allocation – כל תהליך יקבל מספר Frameים לפי גודלו. מספר הframeים עבור תהליך מסוים = (הגודל שלו\גודל כולל של כל התהליכים)*מספר הframeים שיש לנו.

Priority allocation – כמו השיטה הנ"ל רק שאת הגודל נחליף בעדיפות.

אלוקציה גלובאלית - לוקחת frameים שמחזיקים pageים שלא בהכרח שייכים לתהליך שלנו.

אלוקציה מקומית - תיקח בהכרח פריים שמוחזק ע"י התהליך שלנו ברגע שיש page fault.

Thrashing (העמסה) – כאשר לתהליך אין מספיק pageים = יש לו הרבה page fault, יש ניצולת נמוכה של הCPU. בגלל המעבד כמעט ואינו עובד במצב זה, המערכת הפעלה חושבת שהיא צריכה להעלות את **הdegree of multiprogramming**– להוסיף תהליכים, מה שרק יגרום ליותר page fault ואז נגרם thrashing- זהו מצב שבו התהליך עסוק בלעשות Swapים לpageים פנימה והחוצה.
Page fault תמיד שולח אותנו לi/o. כלומר, יש לנו הרבה מאוד i/o , מה שיכול לגרום למעבד להימצא יחסית הרבה זמן במצב idle. איך מערכת הפעלה טובה תנצל זאת?

במצגת יש גרף המציג איך מתנהגת הניצולת של המעבד בהשפעת ה degree of multiprogramming.

Locality – בכל זמן נתון לכל תהליך יש אזורים <u>רציפים</u> בזיכרון שבאמת מעניינים אותו.

ברגע שסכום גדלי הlocality של כל התהליכים > סך גודל הזיכרון הראשי -> thrashing.

הpageים יושבים בזיכרון בצורה רציפה, אך לאו דווקא בframeים רציפים. בתרשים מתוארת הLocality בזיכרון. כל נקודה שחורה מציינת נגיעה של התהליך בpageים. המספרים בצד שמאל מציינים את מספרי הpageים נוצרת צביעה של הנגיעות בpageים השונים לאורך זמן. מגרף זה רואים את עקרון הLocality – תהליכים מבקשים לגעת באותם pageים פחות או יותר בפרקי זמן קבועים. מדוע? כי הפקודות אחת אחרי השנייה פחות או יותר, ולכן מדובר באותו Page, אותם אזורים. חוץ מזה, התהליך עובד על data structure וגם הם שמורים באותם אזורים פחות או יותר בפייג' (למשל מערך). <u>אנו נשאף שכל הpageים שהם חלק מהLocality של התהליך בזמן נתון – יהיו בזיכרון. ואז כמעט ולא יהיו page fault ים.</u>

תהליך הThreshing קורה כאשר סך כל גודל הLocality של התהליכים גדול מסך גודל הזיכרון שיש לי. <u>איך נעקוב אחרי הLocality של התהליך ונעריך את גודלה</u>? (כדי להקצות לכל תהליך זיכרון שהוא לפחות גודל הLocality שלו) ע"י מודל הWorking-Set.

Model Working-Set- נגדיר חלון זמן –דלתא- למשל ביחידות של instructions, שבדוגמה הוא יהיה עשרת אלפים הinstructions האחרונים. נרוץ על הinstructions שהתהליך הריץ, נראה לאן הוא הפנה אותנו, ועבור אותם עשרת אלפים פקודות ("העבר הקרוב"), יעניין אותנו כמה pageים uniqueים (מקוריים) היו בחלון הזמן הזה, וזה יהיה אומדן לגודל הLocality של התהליך בחלון הזמן הנתון. נסמן ב WWSi את הWorking Set (האומדן לגודל ה Locality של התהליך Pi) והוא יהיה סך כל הpageים הuniqueים בזמן הדלתא שהגדרנו. מה צריך להיות גודל דלתא? <u>חלון קטן מדי</u> – יתפוס רק חלק מהLocality, <u>חלון גדול מדי</u>- יתפוס כמה Locality ים. אין פתרון ממשי, אבל כן יודעים את החסרונות במקרים הקיצוניים. אם החלון אינסופי – לקחנו בחשבון את כל הpageים שהתהליך נגע

בהם מהרגע שנפתח עד לרגע שהסתיים. D = סך הדרישה לframeים, אם סך הדרישה יהיה גדול מ-m (=גודל הזיכרון בframeים) – ייווצר Thrashing.

<u>שיטות להקצאת framеים לתהליכים:</u>

אפרוקסימציה - נרצה לעשות שיערוך לWorking Set, איך נעשה זאת? נחזיק מערך שיסמן לנו האם במהלך הזמן שחלף נגענו באותו פייג' מסוים. כל תא במערך מייצג page, ואם נגענו בו נשים בתא הזה 1. בסוף ברגע שנצטרך לפנות איזשהו page מהזיכרון – נדע לקחת מישהו שהוא 0 ולא 1 (פחות מעניין אותנו) וגם נדע מהו גודל הWorking Set (בכל נקודת זמן). כדי שהWorking Set לא יתנפח מדי – נצטרך לאפס את המערך כולו מדי פעם, אחרת יהיה לנו גם Localityים שעבר זמנם. למשל, נאפס את המערך לאחר כל 10000 instructions -> במערך יהיה הlocality עבור ה10000 instructions האחרונים.

<u>נשים לב</u> – אנו בעצם מחזיקים את הworking set שבין 0 ל-10000 הפקודות האחרונות ולא של <u>כל</u> 10000 פקודות אחרונות. למה? בגלל האיפוס שנבצע, אם חלפו רק 200 פקודות יהיה לנו הlocality של ה200 פקודות האחרונות ולא ה10000. עלינו איכשהו לשמור מידע אחורה ולא לאפס לגמרי. נחלק את החלון שלנו ל2 חלונות ובכל חלון נעבוד עם 1 משני מערכים שנשתמש בהם כי לא נרצה בנקודת האיפוס לאבד את כל ההיסטוריה אלא רק חצי ממנה – הישנה יותר.

נאפס 2 מערכים. עבור כל נגיעה בפייג' נמלא את המערך הראשון. כל פעם שנגיע ל5000 פקודות נעזוב את המערך הישן ונתחיל למלא את המערך החדש. כשנגיע לזמן 10000 יש לנו כל ההיסטוריה הדרושה -> נאפס את המערך הישן. ככה בכל פעם יש לנו בכל נקודת זמן היסטוריה של בין 5000-10000 במקום היסטוריה של בין 0-10000.

Page-Fault frequency scheme - נתחזק איזשהו גרף שאומר איך מושפעת כמות ה page faultים כפונקציה של מספר הframeים שמוקצים לתהליך. מה ההתנהגות הtypicallית?

- כשיש מעט framеים – אם נוסיף פריים – כמות הpage faultים תקטן בצורה דרסטית.
- אם יש הרבה framеים ונוסיף פריים – כמעט לא השפענו על כמות הpage faultים.

נגדיר שתי רמות של page fault rate : **upper bound**, **lower bound**. אם הactual rate <u>נמוך</u> מרמת הסף הנמוכה – צריך לקחת ממנו framеים – הוא רווי. אם הוא <u>גבוה</u> – אזי הדרישה לframe נוסף מאוד גבוהה, ויש להוסיף לו frame.

במצגת יש תרשים שמתאר את קצב הpage fault כפונקציה של ציר זמן. בהתחלה התהליך רק התחיל בזמן אפס, והיה Page fault ראשון, וככל שהוא המשיך והתקדם בהרצה – ה page fault rate עלה. עד שיותר ויותר pageים נמצאים כבר בזיכרון מהLocality, ואז הpage fault rate מתחיל לרדת. לאחר מכן הLocality כולו נמצא בזיכרון, ולכן אין כמעט Page faultים. עד שעברנו לLocality חדש, ושוב יש עלייה בPage fault rate, ואז ירידה עד לLocality הבא.

<u>Virtual Memory – Mapped Files:</u>

טיפול בקבצים בצורה יותר מהירה – ע"י מיפוי קובץ לpageים, ושימוש בdemand paging. אם עד היום כדי לגשת לקובץ היינו צריכים לעשות open, read, write, כעת נעשה זאת בדרך יותר מהירה- ניקח את הקובץ שיושב בדיסק, נחלק אותו לוגית לpageים (לבלוקים בגודל pageים), ואז ניקח את הPage table שמייצגת לנו את הזיכרון הווירטואלי של התהליך, ונמפה חלק מהpageים שם, לאותם אזורים בדיסק. כעת המיפוי שלנו יהיה ע"פ הקובץ עצמו. ז"א אם נרצה לגשת לpage מספר 1 של התהליך, ונקבל Page fault, כי הpage עדיין לא נמצא בזיכרון, נביא אותו מהדיסק. מאיפה נביא? בזכות המיפוי נביא אותו מהקובץ עצמו.

נושאים נוספים:

שאלה: איך משפיע גודל הpage על הפרגמנטציה? – תלוי:

פרגמנטציה חיצונית -(יש זיכרון שלא שייך לאף אחד, אבל הוא מפוזר בכל מיני חורים לאורך הזיכרון הפיזי ולכן ייתכן שהגיע תהליך שיכל להיכנס לגודל, אך מכיוון שצריך לשמור אותו בצורה רציפה- לא ניתן לשמור אותו בחורים האלו) – אין השפעה -> זה פותר את הבעיה!

פרגמנטציה פנימית – יש השפעה – מגדיל אותה.

Page table - ככל שגודל הpage יהיה יותר גדול -> כמות הרשומות בpage table תהיה יותר קטנה - > גדול הטבלה יהיה קטן יותר.

I/O overhead - ככל שהpage גדול יותר אנו ניגש פחות לדיסק, מצד שני נצטרך להביא יותר זיכרון. אם הכול יושב בצורה רציפה, אזי הגדלת הpage פי 2 יכולה לחסוך לנו i/o.

TLB Reach – לוקחים רשומות מהpage table שלנו ושמים אותם ברכיב חומרה מהיר. כמות הזיכרון שנוכל לגשת אליה מהר = גודל הטבלה * גודל page. נרצה שכל הLocality שלנו יהיה בTLB.

File System - מערכת קבצים:

קובץ – אוסף (בעל שם) של אינפורמציה שיש בה קשר, שהיא יושבת על אחסון משני (לא הזיכרון הראשי), ולמעשה נוצר לנו פה משהו חדש. הקובץ מייצג לנו רצף של תאי זיכרון (יש חשיבות לסדר).

מה נשמור בקבצים?- data נומרי, או תו, או בינארי, או תכנית שרצה.

מבנה הקובץ:

יש קבצים שאין בהם שום מבנה (אוסף של words או bytes), מנגד יש קבצים בעלי מבנה פשוט למשל של שורות או של נתונים בעלי אורך קבוע, או של נתונים בעלי אורך משתנה (אך יש מפתח להבין מה המבנה הזה אומר). ישנם מבנים יותר מורכבים, כמו למשל Relocatable load file. למעשה ניתן באמצעות הNone לייצג את שתי הדרכים האחרות.

איזה דברים נשמרים עבור קובץ :

- שם הקובץ (תיאור שאמור להיות ברור)
- מזהה חח"ע, שמזהה אותו בתוך מערכת ניהול הקבצים
- סוג הקובץ – אינדיקציה למה שמור בקובץ כדי שנדע איך לקרוא אותו
- מיקום הקובץ – איפה הוא שמור במערכת ניהול הקבצים
- נתוני מיקום – מצביעים לבלוקים עצמם של הdata שיושבים בדיסק
- גודל הקובץ
- נתונים שדואגים להגיד לנו מי יכול לקרוא, לכתוב ולהריץ את הקובץ
- נתונים שעוסקים במתי השתנה הקובץ, מתי נוצר, מי יצר וכו'

כל האינפורמציה הנ"ל נשמרת בדיסק, באיזשהו אזור שאנו קוראים לו **directory** (=מעין structure שמכיל את כל המידע על הקבצים, למרות שהם עצמם לא יושבים שם). גודל הdirectory יכול להיות גדול מאוד.

אילו פעולות נרצה לעשות על קבצים?

1. יצירה Create – מציאת מקום פנוי, וייצור רשומה בdirectory.
2. כתיבה Write – חיפוש בdirectory את הרשומה המתאימה לקובץ המבוקש, ומשם נדע להגיע למקום בדיסק שבו נלך לכתוב את התוכן של אותו קובץ.
3. קריאה Read – אותו דבר.
4. חיפוש בקובץ File seek – הזזת מצביע למקום הבא שנרצה לכתוב או לקרוא מקובץ. נחפש בdirectory את הentry המתאים של אותו קובץ, ואז נשנה את המצביע. זו לא באמת פעולה של i/o.
5. מחיקה delete – חיפוש הקובץ המבוקש, שחרור כל הבלוקים שהיו תפוסים ע"י אותו קובץ בדיסק, ומחיקת הרשומה של אותו קובץ מהdirectory.
6. Truncate – מחיקת הקובץ, אך השארת הרשומה בDirectory (למשל למקרה שנרצה שהמשתמשים יוכלו לכתוב לקבצים, אך לא ליצור אותם).
7. Open(Fi) - חיפוש הרשומה Fi (הקובץ המבוקש) והעברת התוכן של הרשומה לזיכרון (כי אז הפעולות הבאות כנראה יהיו קשורות לקובץ המבוקש)
8. Close(Fi) - העברת התוכן של הentry של אותו קובץ מהזיכרון חזרה אל הדיסק.

באופן כללי נחזיק לכל תהליך את הopen files של אותו תהליך, ונשמור את ה:

1. **file pointer** (דרך פעולת הseek) – עבור כל תהליך שומר את המיקום בקובץ ממנו קראתי או כתבתי לאחרונה.

2. **File-open counter** - כמה תהליכים מחזיקים אותו כקובץ פתוח, למקרה שנגיע לאפס נדע שניתן להוציא את הרשומה מהזיכרון ולהחזיר חזרה לדיסק.
3. נתונים לגבי הdata של הקובץ לשם כתיבה וקריאה.
4. ברמת הקובץ הפתוח – הרשאות הכתיבה והקריאה שלו.

הרבה פעמים נדרוש מנגנון של **Locking** לקבצים שנפתחים (ברגע שמישהו פתח, ננעל כדי שהאחרים לא יוכלו לגשת לשם עיקרון הסינכרוניזציה). יש מ"ה שתומכות במנגנון **Mandatory** ויש שתומכות ב**Advisory**. Mandatory – מניעת תהליכים מפתיחת קובץ שנפתח, Advisory – הודעה לתהליכים שהקובץ כבר נפתח, אך לא מניעה מוחלטת.

File Type – סוג הקובץ - ייוצג ע"י שלוש אותיות אחרונות בסוף שם הקובץ, מצורפת טבלה של דוגמאות במצגת.

איך ניגשים לקבצים? יש שתי גישות לקובץ:

1. **Sequential Access** – ריצה על פני הקובץ רשומה אחר רשומה. כלומר, הפקודות הן read next, write next, reset (הגעה לתחילת הקובץ). למה זה טוב? כי יש לנו הרבה קבצים שזהו הצורה הכי נוחה לגשת אליהם. למשל קומפיילר, כל הזמן קופץ לפקודה הבאה, ולכן זו צורת הקריאה המועדפת עליו. או למשל מעבד תמלילים (הקובץ צריך להיקרא במלואו בסדר הנכון).
2. **Direct Access** – גישה ישירה. נעשה Read n (קריאה מהמקום n בדיסק) , write n, position n (שזה גם וגם). וכך ניתן לממש את השיטה הראשונה בעזרת השנייה (יש טבלה כחולה בהמשך במצגת).

מבנה הdirectory: בשיעור הבא

הרצאה 11 :

Directory – כל המידע על הקבצים שלנו. זהו מבנה נתונים שהוא אוסף של nodeים שמכילים מידע על כל הקבצים – שם הקובץ, גודל הקובץ, הרשאות הגישה לקובץ וכו'.

מנוהל בדיסק, ומכיוון שהדיסק אינו נדיף, נשמור אותו שם פעם אחת ושם הוא יהיה קיים תמיד.

כל רשומה בdirectory, שומרת בנוסף למידע על הקובץ, גם את ההפניות לבלוקים הפיזיים של הקובץ.

איזה פעולות ניתן לעשות על הdirectory? – חיפוש, יצירה, מחיקה, לראות את התוכן - list, לשנות שמות קבצים, לטייל בו. סייר הקבצים מאפשר לנו לטייל בו – הוא קורא את ה directory ומציג לנו אותו.

Partition – מחיצה לוגית שיכולה להשתרע על פני שטח פיזי של דיסק אחד או יותר (או פחות מדיסק). זה בעצם יצירת directory חדש לשמירת קבצים על המחשב שלנו- אנחנו מגדירים איזשהו שטח ליצירת Directory חדש.

יכול להיות שהDirectory עצמו נהרס אבל ה data עצמו עוד קיים.

RAID – מספר דיסקים או מחיצות שמנוהלים בצורה בה המידע מגובה על אחד הדיסקים\המחיצות.

ישנם מקרים בהם לא נרצה לפרמט את הדיסק – במערכות בהן אנו יודעים בדיוק איפה נמצא כל דבר ואין לנו צורך בניהול הקבצים. אנו נחליט איך המידע יישמר על הדיסק, למשל בצורה מעגלית.

נרצה לארגן את ה directory שלנו בצורה הכי טובה – שיהיה קל ומהיר למצוא קובץ וכן שיהיה לנו נוח לתת שמות לקבצים – 2 משתמשים או יותר יוכלו לתת שמות זהים לקבצים, ויהיה ניתן לגשת לאותו קובץ משמות שונים. כמו כן נרצה את היכולת לעשות **grouping** – לאגד קבצים בעלי תכונה מסוימת.

<u>יש לנו מספר תצורות :</u>

- **Single level directory** – יש רמה אחת – directory אחד לכל הקבצים עבור כל המשתמשים. בעיות – לא ניתן לתת שמות זהים לקבצים, לא ניתן לבצע grouping.
- **Two level directory** – שתי רמות - לכל משתמש תהיה תיקייה משלו בה יהיו הקבצים שלו. ניתן לתת שמות זהים לקבצים, החיפוש מהיר יותר.
- **Tree structured directories** – מבנה של עץ, כאשר בכל תיקייה יכולים להיות עוד תיקיות או קבצים וכן הלאה. נוכל לטייל על העץ ולבצע grouping. החיפוש יכול להיות <u>אבסולוטי</u> – מרמת ראש העץ, ביחס לשורש, או <u>רלטיבי</u> – ביחס לתיקייה שאנו נמצאים בה.
- **Acyclic graph directories** – מבנה המאפשר 2 רשומות שמצביעות פיזית על אותו הקובץ או תקייה, כלומר קובץ אחד בעל 2 שמות שונים. הבעיה – אם הקובץ נמחק צריך לוודא שכל המצביעים אליו נמחקים גם כן – ניתן לתחזק על ידי החזקה של Counter.
- **General graph directory –** מבנה הגרף הכללי. יכולים להיווצר לנו מעגלים- בעת מעבר על הdirectory אנו עלולים להיתקע במעגל אינסופי.

Mounting - "חיבור" של מחיצה חיצונית למערכת הקבצים במחשב שלנו.

שיתוף קבצים - ע"י נתינת הרשאות לקובץ, למשתמשים או לקבוצות.

מימוש מערכת הקבצים :

המערכת בנויה משכבות. הרמה הראשונה היא הרמה של התוכנית שלנו שמוציאה בקשות לגישה לקבצים. משם נגיע לשכבה הלוגית – directory – השולחת בקשות לשכבת הorganization שיודעת לחבר את הנתון לבלוק המתאים בקובץ – עושה את הקישור בין הבלוק הלוגי לבלוק הפיזי. משם נגיע לשכבה שמוציאה את הפקודות לdevice driver לכתיבה ולקריאה – Basic file system ואח"כ נגיע ל-i/o שמטפל בקריאה ובכתיבה לדיסק ולבסוף יש את השכבה הפיזית של הDevice עצמו.

איך נמפה בלוק לוגי ולבלוק פיזי על גבי הדיסק?

מושגים :

Boot control block – בלוק שנמצא במחיצה ונותן את כל המידע שהמערכת צריכה על מנת לעשות boot למערכת ההפעלה שמותקנת על המחיצה הזו.

Volume control block – מכיל נתונים על המחיצה עצמה.

Directory – מכיל את המידע על הקבצים, ובתוכו המידע מחולק ל **file control block** שכולל את כל הפרטים אודות הקובץ.

Sector בדיסק = block. ה-sectorים על כל טבעת ממופים למספרים לפי הסדר שלהם מהטבעת הראשונה עד לאחרונה. בין כל 2 סקטורים מתבצע המעבר המהיר ביותר. אנו בעצם לוקחים את הדיסק שלנו והופכים אותו למערך חד ממדי של בלוקים. נשאלת השאלה איך נחלק את הבלוקים השונים לקבצים השונים בdirectory תוך ניצול הדיסק בצורה אופטימלית וגישה מהירה ככל האפשר?

3 שיטות :

- **Contiguous allocation** – כל קובץ יתפוס סט של בלוקים רציפים בדיסק. עבור כל קובץ נשמור את בלוק ההתחלה וכמות הבלוקים שיש לנו עבור אותו הקובץ. מצד אחד השיטה הזו מאפשרת לנו **random access** – גישה מהירה לקבצים שלנו, פשוטה למימוש אך מצד שני השיטה בזבזנית ויוצרת פרגמנטציה – ניתן להתמודד ע"י דחיסה – איחד כל החורים לחור אחד גדול (מאוד בזבזני). כמו כן קבצים לא יכולים לגדול אם אין בסופם בלוקים פנויים.
הרבה פעמים מכריזים על כמות קבועה של בלוקים לקובץ אך בפועל מקבצים כמה בלוקים יותר על מנת לאפשר לקבצים שיצטרכו אולי בעתיד לגדול. זה גם גורם לפרגמנטציה.
- **Linked allocation** – ניקח כל בלוק כאשר נבזבז מתוכו טיפה מהמקום לטובת פוינטר שיצביע לבלוק הבא של אותו הקובץ. כך נצטרך רק את כתובת הבלוק הראשון ודרכו נגיע לבאים. בשיטה זו אין לנו random access – עלינו להתחיל מהבלוק הראשון עד שנגיע לבלוק שנמצא באמצע או בסוף. אם הלך לנו מצביע – הלך לנו כל הקובץ. הרבה פעמים משתמשים בשיטה זו ב- FAT– שיטה בה לוקחים את כל המצביעים בבלוקים ומעבירים אותם לזיכרון – main memory, שם נקצה טבלה רציפה שגודלה כמספר הבלוקים בדיסק ולתוכה נקרא את כל המצביעים של הבלוקים השונים. זה משמש אותנו לצורך גישה מהירה יותר.
- **Index allocation** – עבור כל קובץ נשמור בלוק שלם שיחזיק בתוכו את המצביעים לכל הבלוקים האחרים של אותו הקובץ – בלוק האינדקסים. אם יש לנו קובץ מאוד גדול תהיה לנו בעיה לשמור בבלוק הזה את כל האינדקסים, לכן נוכל לשרשר באינדקס האחרון מצביע לבלוק נוסף שיאחסן בתוכו את האינדקסים הבאים. השיטה הזו מאפשרת לנו **random access.**

עלינו לנהל את הבלוקים הפנויים בדיסק. השיטה הנאיבית אומרת לשמור ווקטור בינארי של ביטים כמספר הבלוקים בדיסק, כאשר 0 מסמן בלוק פנוי ו-1 מסמן בלוק תפוס. ברגע שנצטרך להקצות בלוק חדש, נחפש את התא הראשון בווקטור שיש בו 0 ונקצה את הבלוק המתאים לו.

כל בקשה לנגיעה בקובץ תעבור דרך log של הדיסק עצמו, הבקשה תירשם בלוג. הבקשות יהיו מסודרות בו לפני סדר ההגעה שלהן. אם הייתה תקלה והמערכת נפלה, נוכל לחזור ללוג ולאחר התאוששות המערכת, להריץ את כל הבקשות שהתקבלו לפני הנפילה, כך נוכל להחזיר את מערכת למצב הכי עדכני ונוכל להיות בטוחים שכל הבקשות שהגשנו התבצעו.

Disk scheduling :

Access time – זמן הגישה למידע על הדיסק. מורכב מ- seek time, rotational time.

Seek time – זמן החיפוש של הראש הקורא הכותב של המידע על הדיסק. זה הזמן שלוקח לראש הקורא כותב להתמקם על הטבעת שבתוכה הSector הרצוי. זהו המרכיב הדומיננטי בזמן הגישה.

Rotational time – זמן הסיבוב של הדיסק – הזמן שלוקח לראש להגיע לתחילת הSector בטבעת.

נרצה להגיע למינימום של seek time.

יש מספר אלגוריתמים שמגדירים לראש הקורא כותב לאן לזוז בזמן קריאת המידע מהדיסק :

- **FCFS** – בהינתן תור של בקשות לקריאה מסקטורים מסוימים, נטפל בבקשות לפי סדר הגעתן.
- **SSTF – shorter seek time first** – נבחר לטפל בבקשה שהכי קרובה לראש הקורא הכותב בכל רגע.
  החיסרון – מי שמקבל בדרך כלל את השירות הם הסקטורים שנמצאים במרכז -> יוצר "הרעבה" כלפי הסטורים הקיצוניים.
- **Scan** – נעבור מקצה אחד לשני כאשר נשרת בדרך את כל הסקטורים הרלוונטיים, ברגע שהגענו לקצה אחד נחזור לקצה השני.
  החיסרון – הסקטורים במרכז מקבלים יותר התייחסות מאשר הסקטורים בקצוות.
- **C-Scan** – כמו השיטה הנ"ל, רק שבחזור לא ניתן שירות לאף אחד. בשיטה זו המרכז מאבד את היתרון שלו.
- **C-Look** – כמו השיטה הנ"ל, רק שהוא לא מגיע לקצוות במידה ואין צורך אלא נעצר בסקטורים הקיצוניים ביותר שדורשים טיפול.
  החיסרון – כאשר הגענו לסקטור הקיצוני ביותר והתחלנו לחזור לצד השני, אם בזמן הזה יגיע סקטור קיצוני יותר, יהיה עליו לחכות זמן רב עד שנחזור אליו.

הביצועים תלויים במספר הבקשות שמגיעות.

האלגוריתמים הכי טובים הם – SSTF, C-Look.