מערכות הפעלה

מרצה

פרופ' דרור פייטלסון

מתרגל | עידן רפאלי

דניאל דייצ'ב

דויד קיסר שמידט

deychev.com

מערכות הפעלה | 67808

נכתב ע"י דויד קיסר-שמידט ודניאל דייצ'ב

23 ביוני 2022

מרצה | דרור פייטלסון ודוד חי

מתרגל | תומר מיכאל, עידן רפאלי

מצאתם שגיאה? ספרו לנו! שלחו לנו מייל לאחת מהכתובות שכאן:
daniel.deychev@mail.huji.ac.il
david.keisarschm@mail.huji.ac.il

תוכן העניינים

רשימת אלגוריתמים

Listings

סיכום זה מוקדש לדוידניאל.

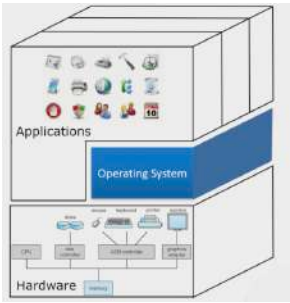
מערכות הפעלה! מלא בשמחה, אהבה וגלידה ועומד להיות לנו כיף אז תתכוננו D:

יש לנו הרבה נושאים, ביניהם: כל מיני!

Lets jump right into it!

1 מבוא

מהי מערכת הפעלה? במחשב שלנו יש כמה רכיבים: אפליקציות, רכיבי חומרה (עכבר, מקלדת, מסך...), זכרון, מעבד ועוד. כדי שהאפליקציות יוכלו לרוץ, מערכת ההפעלה צריכה לקשר ביניהן לבין החומרה. יחד עם זאת, בפועל אפליקציות לא צריכות לתקשר עם מערכת ההפעלה כדי לגעת במעבד (CPU), אלא ברכיבים חיצוניים כמו העכבר והמקלדת, וזה נעשה באמצעות פסיקות חומרה (interapts). בנוסף, מערכת ההפעלה איננה מאפשרת לאפליקציה אחת בלבד לרוץ בכל רגע נתון, אלא יכולה לאפשר לכמה אפליקציות וכאשר אפליקציה רצה, מערכת ההפעלה לא רצה, אלא אם הייתה קריאה אליה, או פסיקת שעון - רכיבים חיצוניים שקוראים לה. נוכל לחלק את רכיבי המחשב לפי האיור הבא:



איור 1: רכיבי המחשב. שכבת חומרה אחת לכולם, מערכת הפעלה אחת לכולם המקשרת בין הרכיבים, וכמה אפליקציות שונות שרצות במקביל.

עתה, נשאלת השאלה, מה מגדיר את מערכת ההפעלה? למעשה יש שני ממשקים מרכזיים.

- הממשק שמגדיר את החומרה הוא הארכיטקטורה של המחשב. למשל, ארכיטקטורת Von Neumann. המשמעות היא אילו פקודות המחשב מסוגל לעשות (JMP, ADD, XOR, NAND ...), והיא כוללת גם את קידוד הפקודות לביטים.
- הממשק שמגדיר את מערכת ההפעלה הוא קריאות המערכת (System Calls), כלומר, הקריאות לגישה אל רכיבי החומרה.

כאשר מערכת ההפעלה רצה, היא צריכה לתזמן הרצת אפליקציות שונות במחשב. למשל, באופן כללי, היא מבצעת את ההליך הבא:

1. כאשר היא מחליטה להריץ אפליקציה, היא מקצה עבורה את המעבד.
2. במהלך הריצה, שעון המערכת שולח פסיקה (בדרך כלל $100 - 1000$ פעמים בשנייה).
3. כאשר זה קורה, המעבד מריץ את הרכיב שמטפל בפסיקות השעון, אשר מחליט איזו אפליקציה להריץ.
4. עתה האפליקציה שרצה שולחת פסיקת קריאה למערכת ההפעלה כדי לגשת לרכיב חומרה.
5. הקריאה יוצרת trap, וגורמת למערכת ההפעלה להריץ את הרכיב שהתבקש.

כל אחד מהשלבים משתמש בתוכנה כתובה, וסופית. בשונה מקומפיילר ותוכנה שאנו כותבים, מערכת ההפעלה תמיד רצה, ולא מפסיקה לרוץ, אם היא עוצרת, משהו לא בסדר.

תפקידיה של מערכת ההפעלה הם מתן **הפשטה ווירטואליזציה** למשתמשים, מה שנוח למשתמש.

כמו כן, עליה לבצע הכל בצורה יעילה, **ולנהל משאבים** בצורה מיטבית.

1.1 הפשטה (Abstraction)

כמה מערכת ההפעלה חיונית עבורנו? נביט למשל בקוד הבא:

```
void foo(){
        return_code = read(fd, buffer, size);
}
```

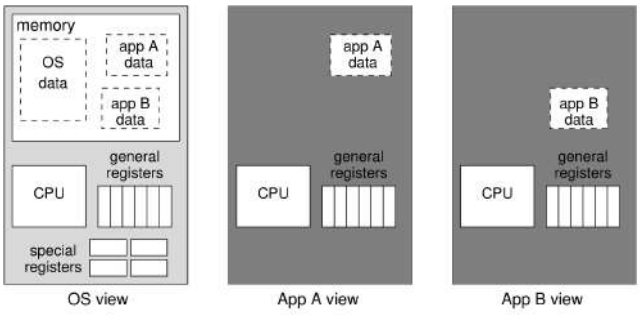
אנו קוראים מידע בגודל $size$ בייטים מהקובץ fd אל תוך $buffer$. זהו ממשק נוח (יחסית) לשימוש, אך הוא עושה שימוש נרחב במערכת ההפעלה - הפשטה. השימוש בשפה גבוהה ולא בשפת נמוכה מתאפשר בעיקר בזכות מערכת ההפעלה, שכן בזכותה אנחנו יכולים לקרוא לקומפיילר שיתרגם את הקוד לשפת מכונה. על כן, יהיה עלינו לכתוב הכל בשפת מכונה. מעבר לכך, נהיה צריכים לשמור בכל פעם את המיקום הנוכחי שלנו בקובץ ולשלוח ל-CPU. יתר על כן, כיצד נדע לאיזה קובץ לגשת? עלינו לקרוא את הקובץ מהדיסק, אך זה אומר שעלינו לגשת למיקום ספציפי בדיסק, שכולל - מספר המשטח, והחלק שמתאים לו בתוך המשטח. זה מאוד לא נוח. על כן, נסיק כי מערכת ההפעלה היא חיונית על מנת לאפשר לנו שימוש נוח במחשב. בנוסף, נניח שנרצה לגשת לרכיבים חיצוניים כמו מקלדת ועכבר - יהיה עלינו לעשות זאת בכוחות עצמנו בכל פעם (אפילו בכל הקלדה על המקלדת), ועל

אף שיש ביכולתנו מחשב שלם, ולכן תאורטית אנחנו מסוגלים לעשות הכל, אנחנו מהר מאוד נתייאש. זאת ועוד, הרצת כמה אפליקציות במקביל זה דבר שלא ברור בכלל איך לבצע.

בפועל, למחשבים שלנו יש יותר ממעבד אחד, ולכן כל תהליך הגישור מסובך הרבה יותר ודורש שמירה של מצב המעבדים בכל רגע נתון. יתכנו גם מחשבים עם כמה מערכות הפעלה, על ידי שימוש בוירטואליזציה.

לבסוף, אנו מבינים כי מערכת ההפעלה מספקת לנו חווית משתמש נוחה, ללא התעסקות בסיבוכים שהיא מתעסקת בהם, כמו טיפול בחומרה. אחד הדברים המופלאים שהמערכת מספקת לנו היא "קובץ". כמו שהזכרנו קודם, קובץ נשמר בבלוק בזכרון הדיסק, אך מערכת ההפעלה מאפשרת לנו לגשת לקבצים לפי שמות, ואף לתת לקבצים שונים שמות זהים, אם הם בתיקיות שונים, על ידי יצירת עץ תיקיות, וסיפוק מבנה היררכי למשתמש.

כמו כן, ראינו שלמערכת ההפעלה חשיבות מכרעת בהרצת אפליקציות, ובעוד היא רואה את כל רכיבי המחשב - חומרה, מעבד, זכרון, ובפרט זכרון האפליקציות, האפליקציות מכירות אך ורק את המעבד והזכרון - הן לא מודעות אחת לשנייה ולכן, אם מערכת ההפעלה תטפל בהן כנדרש, הן לא יפריעו אחת לשנייה. כלומר, מערכת ההפעלה אחראית לבידוד האפליקציות השונות במחשב:

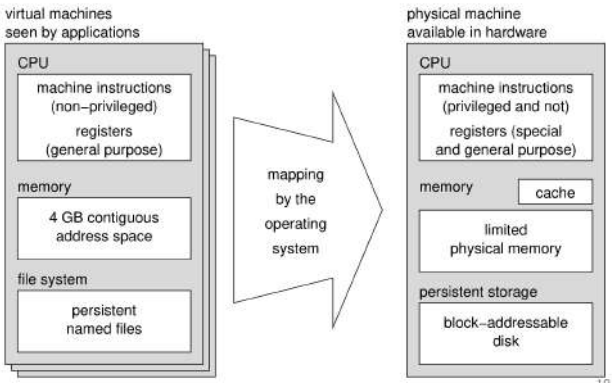


איור 2: המחשה לראייה ה"צרה" והמופשטת של האפליקציות לעומת ראייתה הרחבה של מערכת ההפעלה

אז מערכת ההפעלה מספקת לנו ממשק נוח יותר ויוצרת מה שאין בחומרה, אך מתי הדבר יהווה בעיה? הוא מונע מאיתנו לבצע אופטימיזציות. למשל, דיסק הזכרון ממנו קוראים מידע מסתובב, ובכל פעם קוראים ממנו מידע, אם לא תהיה לנו גישה ישירה אליו לא נוכל לקרוא מעגל שלם מהדיסק בפעם אחת, במקום לקרוא חלק מהמעגל, לתת לו להסתובב, וכך הלאה עד שסיימנו לקרוא את כל המעגל. בפועל המחשבים שלנו משתמשים היום ב-(Solid State Disk) SSD שלא באמת מסתובב.

1.2 וירטואליזציה (Virtualization)

וריטואליזציה הוא שם להתנהגות של מערכת ההפעלה - היא יוצרת אשליה לתהליכים שקורים. למשל, אפליקציות חושבות שהן היחידות שרצות, וההרצה שלהן במקביל גורמת לנו לחשוב שהן רצות במקביל, למרות שבפועל הן לא. כלומר, תפקידה של מערכת ההפעלה היא לתרגם את האשליה שחווה כל יישות במחשב (למשל אפליקציה) למה שקורה בפועל.



איור 3: בעוד האפליקציה רואה קבצים כגורמים בעלי שמות, מתרגמת מערכת ההפעלה אותם לבלוקים בעלי כתובת בדיסק. על אף שהיא רואה 4GB זכרון, בפועל מערכת ההפעלה יכולה להקצות לה הרבה פחות וכך לאפשר להרבה מאוד אפליקציות לראות אותו מרחב זכרון, אך בפועל לגשת למרחב מצומצם.

על אף שוירטואלזיציה נראית כשקולה לתוספים והפשטה כשקולה לשיפור השימוש, ההפרדה בין שני המושגים איננה חד משמעית, ולרוב המכונה שמיוצגת על ידי מערכת ההפעלה היא גם מופשטת וגם וירטואלית. מלבד זאת, וירטואליזציה הוא מושג בפני עצמו מעבר למערכות הפעלה.

1.3 ניצול משאבים

מערכת ההפעלה צריכה לנצל משאבים בצורה מיטבית, על ידי התחשבות ביעילות, ביצוע והוגנות. על כן אנו מגדירים גדלים לאומדן הניצול.

הגדרה. זמן תגובה/השהייה (Latency/Response Time) הוא הזמן שלוקח לבצע פעולה או משימה. נמדד בשניות.

הגדרה. תפוקה (Throughput) הוא קצב ביצוע העבודה. למשל, מספר הפעולות שמבוצע בשנייה. נמדד ב-sec^{-1}.

הגדרה. ניצולת (Utilization) הוא הזמן היחסי שהמחשב עובד ולא נח, כלומר מנצל את משאבי החישוב.

הגדרה. תקורה (Overhead) גישה למשאבים ישירים או לא. למשל, גישה לזכרון - קריאה מהזכרון איטית בהרבה מפעולת מעבד.

מסקנה. מתקיים כי $\text{Overhead} + \text{Utilization} + \text{idle Time} = 100\%$ שכן חלק המערכת או לא עושה כלום אומאפשרת למכשיר לעבוד, או מחכה לגישה למשאבים.

1.4 Multi Programming

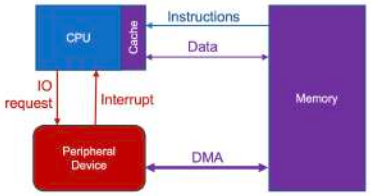
כפי שראינו קודם, ללא מערכת הפעלה, לא ברור איך לבצע כמה משימות במקביל. אך עם מערכת הפעלה, משימות מתבצעות במקביל. דוגמא מוכרת היא ביצוע תכנית אחת, תוך כדי הדפסה למסך של תכנית אחרת על ידי שליחת הבקשה להדפסה ל-Controler, שנקרא $\text{Printer} - \text{Spooler}$.

אנו מניחים:

1. המעבד תמיד שם ואי שימוש בו הוא בזבוז.
2. המעבד מהיר בהרבה מה-I/O.
3. יש גישה ישירה לזכרון ב-DMA - כדי שהפריפריאלים יוכלו לעבוד במקביל מבלי להפריע למחשב.
4. יש יותר ממשימה אחת לביצוע ביחידת זמן.

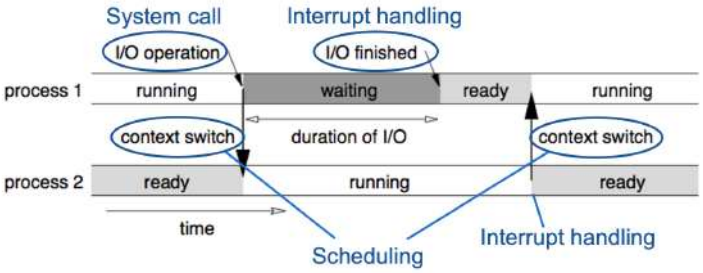
על כן, מערכת ההפעלה צריכה:

1. מערכת I/O.
2. ניהול זכרון ושמירה עליו.
3. מתזמן - קובע מי משתמש במעבד.
4. Spooler - שולט ברכיבי I/O איטיים יותר.



איור 4: רכיב פריפריאלי מסוגל לגשת ישירות לזכרון על מנת לחסוך זמן למעבד. כאשר המעבד רוצה לגשת לרכיב כנ"ל הוא שולח בקשת I/O. מכיוון שקריאה מהזכרון זו פעולה יקרה עבור המעבד, הוא שומר מאגר "נגיש" יותר Cache.

בהרצה במקביל של תכניות, מערכת ההפעלה אחראית על חלוקת המשאבים ותזמונם:



איור 5: כאשר קריאת ה-I/O לרכיב פריפריאלי מתבצעת, מתחילה תכנית 2 לרוץ ותוך כדי זה, רכיב ה-I/O "נטען" עבור תכנית 1. לאחר מכן באמצעות פסיקה, מערכת ההפעלה ממשיכה את ריצתה של תכנית 1.

למעשה, תפקידיה המרכזיים של מערכת ההפעלה הם (נרחיב על המושגים בהמשך):

1. ביצוע תכניות - יצירת תהליכים.
2. ניהול תהליכים - והריגתם בעת סיומם.
3. ניהול זכרון - להקצות לכל אפליקציה זכרון, לא יתר על המידה.
4. מערכת הקבצים.
5. Networking - הרצת פרוטוקולי התקשורת.
6. בטחון ושמירה - מניעת גישה לקבצים רגישים או פרטיים.
7. הקצאת משאבים ומניית המשאבים.
8. ניהול הרכיבים.
9. תקשורת עם המחשב - Shell.

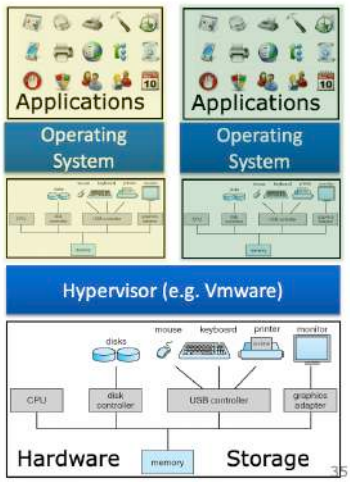
הערה. מבחינת התפתחות המעבדים, ב-50 השנים האחרונות הייתה עלייה אקספוננציאלית של מספר הטרנזיסטורים במעבד. ככל שיש יותר מהם, ככה החישובים של המעבד לוקחים פחות זמן. בקרוב העלייה תקטן, כיוון שהם מתקרבים לגדלים אטומים. למעשה, קצב השעון הפסיק לעלות אקספוננציאלית ב-2005, וזאת מכיוון שבגודל מסוים, יש סכנה שהטרנסיטורים ישרפו, כיוון שאינם יכולים לפלוט את החום. לכן מאז 2005 הקצב נשאר קבוע וכך גם האנרגיה הדרושה עבור מחשב אינה עולה יותר מדי. באותה השנה החלו לעלות מספר הליבות של מחשב, וכך על אף שקצב השעון לא עולה, עולה מהירות המחשב - מה שדורש תמיכה של מערכת ההפעלה בתכנות מקבילי אמיתי ($Multi-Core$ Programming).

יש שתי גישות של תכנות מקבילי:

- Multi Programming - יש מעבד אחד, ותהליכים מתחלפים כאשר יש הוראה שדורשת המתנה, כמו המתנה לקלט מהמשתמש, קריאה מהזכרון וכו'.
- Time Sharing - יש מעבד אחד, ותהליכים מתחלפים כמו בשיטה הקודמת. אבל, אם מערכת ההפעלה מאמינה שהחלפה תשבר את הניצולת, היא תבצע החלפה כרצונה - מה שאומר שיהיו יותר context switches בפועל.
- $Multi-Core$ - תהליכים רצים במקביל על מעבדים שונים ומערכת ההפעלה מתזמנת אותם על כל מעבד כרצונה.

דוגמאות לוירטואליזציה

1. נוכל ליצור חלוקה של דיסק קשיח. כך שכל בלוק בחלוקה יהיה דיסק בפני עצמו, ולמרות שבפועל מדובר בדיסק אחד, מערכת הפעלה תיצור את האשליה שמדובר בכמה דיסקים שונים. בדיסקים מתקדמים אפשר לעשות את הוירטואליזציה מתחת למערכת ההפעלה.
2. Virtual Private Network(VPN) נוכל ליצור חיבור אינטרנטי לאזור אחר בעולם באמצעות מערכת ההפעלה.
3. נוכל ליצור וירטואליזציה לחומרה על ידי הוספת שכבת Hypervisor (e.g Vmware) מעל החומרה. ככה נוכל להריץ מערכות הפעלה נוספות תחת מערכת ההפעלה המרכזית של המחשב שלנו. הם יגשו לחומרה הוירטואלית שתנוהל בפועל על ידי מערכת ההפעלה שלנו.
4. אפליקציה שיוצרת וירטואליזציה ומעליה אפליקציות שרצות על מערכות הפעלה אורחות.
5. קונטיינרים - אריזה של האפליקציה עם כל הסביבה שהיא צריכה כדי לרוץ, (ללא וירטואליזציה של החומרה) למשל Docker[1]



איור 6: המחשה לוירטואליזציה של החומרה

[1] משתמשי mac וודאי מבינים את חשיבותו עבור שימוש ב-Valgrind.

יתרונות הוירטואליזציה

1. מאפשרת מגוון רחב של מערכות הפעלה.
2. שימוש במכונות וירטואליות - אותן ניתן להקפיא ולהפשיר בעת הצורך.
3. העברת מכונה וירטואלית לבעלים אחרים.
4. ייצור מכונות וירטואלים ומחיקתן כרצוננו.
5. בידוד מערכות וירטואליות - מוסיף ביטחון.

כמובן, הוירטואליזציה פוגעת ביעילות.

1.5 הענן

כיוון שמשאבי חישוב חיוניים ליעילות המערכת, משתמשים הרבה במשאבי חישוב על ידי שימוש באינטרנט, למשל שומרים את מכונות החישוב עצמן במקומות בהם החשמל זול, ומתקשרים איתם מרחוק. כלומר עבור אפליקציה צריך דפדפן אחד, במקום מחשב שלם. כמו כן, באמצעות וירטואליזציה אפשר לחלק את המשאבים בשרת אחד - יצירת הרבה מכונות וירטואליות עליו.

הערה. הטלפונים הסלולרים שלנו עובדים מעט שונה - כאשר המעבד לא עובד, אין פירושו בזבוז, אלא חסכון בבטריה, ולכן ההנחות שלנו צריכות להתעדכן: אין בהכרח יותר ממשימה אחת לביצוע, כמות הזכרון אינה גדולה מספיק בהכרח.

2 מצב גרעין ופסיקות (Kernel Mode & Interrupts)

2.1 מצב גרעין, מצב משתמש ומעבר ביניהם (Kernel Mode & User Mode)

למחשב שלנו יש שני סוגי פקודות:

1. פקודות ללא פריבילגיות (לא מיוחסות): כל הפקודות האריתמטיות, ופקודות פשוטות של המשתמש.
2. פקודות עם פריבליגיות (מיוחסות): פקודות מיוחדות למערכת ההפעלה. אנחנו לא רוצים שהמשתמש יוכל להשתמש בהן, ולכן מערכת ההפעלה בודקת האם למשתמש יש גישה לפקודה שהוא מנסה להריץ.

דוגמה. נניח שהמשתמש רוצה לקחת בלוק של זכרון מהדיסק (כלומר לקרוא קובץ, זו הרי אבסטרקטיזציה של ה-OS). מערכת ההפעלה תבדוק שיש לו את ההרשאות לקריאת הקובץ.

הגדרה. מצב גרעין (Kernel Mode). זהו מצב ריצה של מערכת ההפעלה. במצב זה, המשתמש יכול להריץ כל פקודה של המעבד - מיוחסת או לא מיוחסת.

הגדרה. מצב משתמש (User Mode). מצב ריצה של המשתמש. המשתמש יכול להריץ אך ורק פקודות לא מיוחסות על המעבד.

שאלה כיצד נממש Kernel/User Mode?

תשובה למעבד יש רג'יסטר מיוחד שנקרא PSW – (Processor Status Word). ברג'יסטר יש ביט שמסמן את המצב (Kernel/User), ונקרא ה-ModeBit. כאשר הביט מכיל User, פקודות מיוחסות לא יעבדו. יחד עם זאת, מעבדים מודרנים מאפשרים יותר משני מצבים, כאשר כל מצב מכיל פריבליגיות שונות (חזקות יותר או חלשות), לכן במעבדים אלה יש $2-4$ ביטים למצב. למשל במעבד ARM יש 7 מצבים ולכן 3 ביטים.

למעשה, כיוון שמצב הגרעין מאפשר הרצה של כל פקודה אפשרית, נרצה להבטיח שרק מערכת ההפעלה רצה במצב גרעין, ושכל שאר הגורמים ירוצו במצב משתמש. מכאן עלינו לקבוע מנגנון לכניסה למצב גרעין. יש 3 דרכים:

2.1.1 System Calls

מערכת ההפעלה נותנת שירותים למשתמש באמצעות קריאות אלה ומכניסה אותו למצב Kernel Mode בצורה כביכול "מפוקחת".

הקריאות הנ"ל יכולות להיקרא על ידי המשתמש כמו פונקציה מספריה, רק שבפועל, פונקציה זו היא ממערכת ההפעלה ורצה במצב הגרעין.

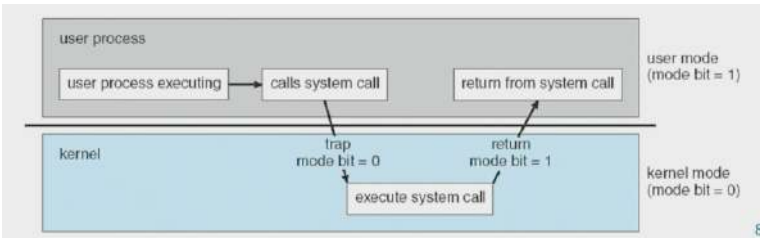
כמו כן, המשתמש אינו מסוגל לגשת ישירות למבנים פנימיים של מערכת ההפעלה, אלא רק דרך SystemCall.

הערה. קריאות אלה עושות ואלידציה לארגומנטים שהן מקבלות במטרה למנוע פרצות אבטחה.

על מנת לבצע את המעבר בין שני המצבים, מערכת ההפעלה צריכה לספק למשתמש פקודת מעבר, שהינה פקודה לא מיוחסת (אחרת לא היה יכול להריץ אותה). פקודה זו נקראת trap. הפקודה מבצעת את הדברים הבאים:

1. משנה את ה-mode bit למצב גרעין.
2. טוענת ל-PC את הכתובת לפונקציית ה-OS שביקשנו ב-System Call, ומוודאת שזו אכן פונקציה של מערכת ההפעלה. זה מעין קסם שהיא מסוגלת לעשות, עליו נעמיק בהמשך.

פקודה זו נקראת באופן הבא:

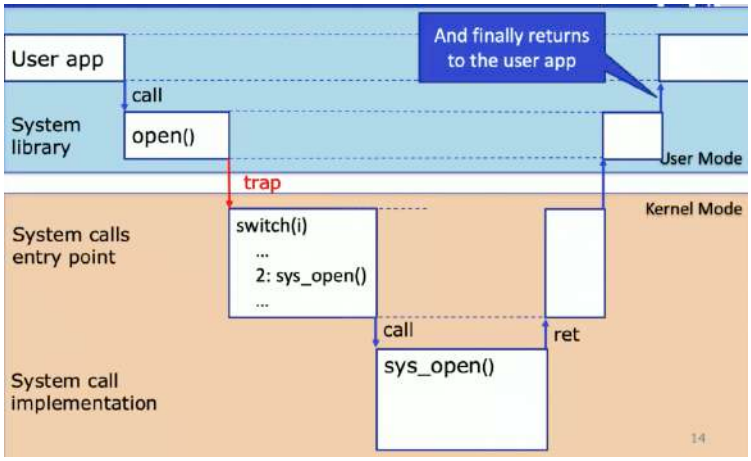


איור 7: נניח שהתכנית של המשתמש רצה והוא קורא בתוכה למערכת ההפעלה:

User Process Executing → Calls System Call

בתגובה, תרוץ הפקודה trap ותגדיר mode bit = 0. רק לאחר מכן תרוץ הפונקציה שביקשנו. כך הכניסה מערכת ההפעלה את המשתמש אל תוך ה-kernel mode. עתה הקריאה תתבצע בהצלחה, ובסיומה יוחזר ה-mode bit = 1 למצבו המקורי, ובכך יחזור המשתמש למצב User.

פקודת ה-trap בהנתן תכנית רצה של המשתמש, נניח שהוא ביצע קריאה למערכת ההפעלה. למשל, לפונקציה Open על מנת לפתוח קובץ. כדי לבצע את הקריאה, על המשתמש לספק את הפרמטרים הרלוונטים לפונקציה, למשל, שם הקובץ לפתיחה ומצב הפתיחה, אך גם את המזהה של ה-System Call אותו מערכת ההפעלה תריץ. למעשה, למערכת ההפעלה יש טבלה שלמה של System Calls עם מזהים (סדר גודל של מאות).
לאחר שהמשתמש קרא למערכת ההפעלה בדרך זו, מתבצעת פקודת ה-Trap שנקראת עקב הקריאה למערכת ההפעלה. פקודה זו, כפי שתואר קודם, תשנה את ביט המצב למצב קרנל ותקרא לפונקציה שהמשתמש בחר בה. מכאן תקרא פונקציה של מערכת ההפעלה שאחראית לניהול כל קריאות המערכת. מה שיש שם הוא למעשה Switch אחד גדול, שלפי קוד הקריאה, יקרא ל-System Call הרלוונטי, עם הפרמטרים הרלוונטים. מכאן תרוץ הפונקציה שרצינו, sys_open.
עתה, נותר לחזור חזרה מה-System call עם פקודת reutrn והחזרה של כל הארגומנטים הרלוונטים - האם הפעולה הצליחה (כשלון מוגדר כ-0 בדרך כלל), פוינטר לקובץ שנפתח וכמובן, מחזירה חזרה את הכתובת של ה-PC לכתובת הפונקציה שקראה ל-System Call. כל זה קורה בכמה שלבי Return, כמפורט בהליך הבא:



איור 8: ניתן לראות את שלבי הקריאה השונים, את תפקיד ה-trap המקשר בין שני המצבים ואת שלבי החזרה. כל התהליך יקר מבחינת תקורהת שכן יש כן גישה למשאבים לא ישירים כמו הדיסק.

יש כל מיני סוגים של System Calls. ביניהם - ניהול קבצים, תקשורת (בין מחשבים), שליטה על תהליכים, מכשירים חיצוניים (מקלדת, עכבר...), תחזוקת המידע, אבטחה (הרשאות גישה).

2.1.2 Exceptions

פקודת Exception (חריגה) נגרמות כאשר מעבד "לא יכול" לבצע פקודה.

שאלה מתי מעבד "לא יכול" לבצע פקודה? הרי אין לא יכול, יש "לא רוצה".

תשובה למשל, כאשר הפונקציה היא ב-Kernel Mode, אבל המשתמש מנסה להריץ אותה ישירות, אך יש עוד הרבה מקרים:

1. במעבדים של 64 ביט, אין בהכרח 2^{64} פקודות חוקיות, לכן כאשר המעבד מקבל פקודה לא חוקית, הוא שולח Exception.
2. שגיאת זמן ריצה - למשל חילקנו באפס.
3. שגיאת תכנות של המתכנת (באסמבלי) - אין פריבליגיות, או פקודה לא חוקית.
4. גישה לא חוקית לזכרון (Segmentation Violation).

שאלה נניח שהמעבד קיבל פקודה לא חוקית, מה הוא אמור לעשות?

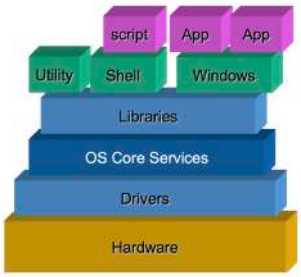
תשובה המעבד רץ בקצב פנטסטי, אין לו הרשאות, הוא רק מריץ. אם הפקודה לא חוקית, זו לא בעיה שלו, אלא של התוכנה, שכן הבאג בתוכנה, כלומר של מערכת ההפעלה. בפרט, החומרה תזרוק את האחריות על מערכת ההפעלה ותריץ את הפונקציה המתאימה שתטפל בשגיאה הנ"ל.

שאלה מערכת ההפעלה מגלה מהמעבד על הפקודה הלא חוקית. מה עליה לעשות?

תשובה בדרך כלל אין יותר מדי מה לעשות מלבד להרוג את התהליך שרץ (המונח "תהליך" יובהר בהמשך הקורס). לכן היא שולחת סיגנל לתוכנה שהורג את התהליך, אותו ניתן לתפוס במנגנוני Try/Catch, אך אם הוא לא נתפס, הוא חוזר למערכת ההפעלה והתהליך מת לחלוטין. יחד עם זאת, יש מקרה מיוחד שבו מערכת ההפעלה מסוגלת לטפל - ניהול זכרון.

2.1.3 שכבות מבנה המחשב

לא תמיד ברור מה האחריות של מערכת ההפעלה. באופן כללי, המבנה של המחשב מעט מורכב יותר ממה שתיארנו בהתחלה וכולל שכבות רבות, אותן נתאר בקצרה:



איור 9: שכבות מבנה המחשב

1. — שכבת ה-OS Core Services כוללת מגוון רחב של שירותים. האם החלונות וניהול המסך הם חלק משירותים אלה? תלוי במערכת ההפעלה. למשל, Windows קרויה על שם כך שהיא אחראית לזה, אבל מערכות הפעלה אחרות כמו Unix/Linux, יש אפליקציה חיצונית שאחראית לכך והיא נמנית בשכבה הירוקה (Windows). למעשה מייקורספוט ניסתה בעבר להכריח כל חברה לכלול גם ניהול חלונות וגם דפדפן בתוך מערכת ההפעלה, ומניעת התקנה של דפדפנים חיצוניים. בית המשפט פסק שהדבר אינו חוקי, למזלנו.

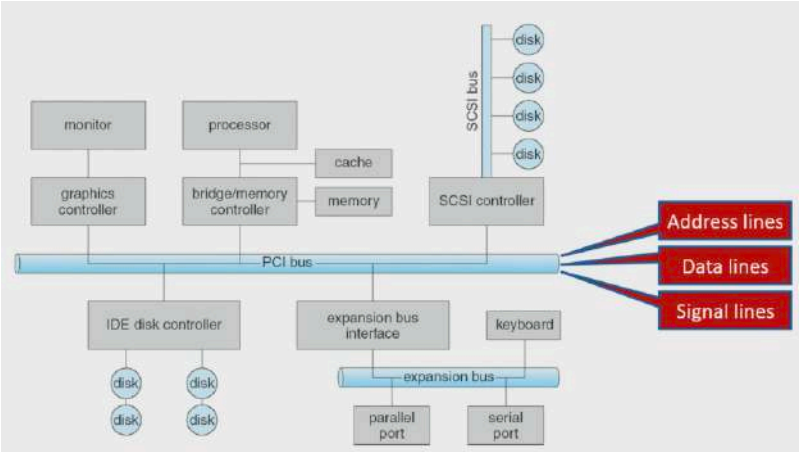
2. — ה-Shell בשכבה הירוקה אחראי על הרצה של התכנית שאנו כותבים, במצב משתמש. למשל הפקודה echo (254 שורות מימוש ב-c) שקוראת ל-System Call שקורא מקובץ - Write.

3. — שכבת הספריות, מכילה קבצי ספריות שאנו מייבאים בעת כתיבת תכניות, והן תלויות בשפת התכנות, למשל libc, libc++...

4. — שכבת שירותי מערכת ההפעלה היא השכבה שבה הקורס שלנו עוסק בעיקרו - אכסון, תהליכים, ותקשורת.

5. — שכבת ה-Drivers היא התקשורת של מערכת ההפעלה עם החומרה. לכל רכיב אפשרי יש Driver ולכן יש הרבה מאוד Drivers. למשל HP Disk פועל שונה מ-Apple Disk, וה-Driver הוא זה שיודע לתפעל אותו. לכן מערכת ההפעלה מריצה את ה-Drivers. ה-Drivers רצים במצב Kernel כדי שלא יצטרכו ליצור עוד קשר עם מערכת ההפעלה. דבר זה מסוכן, והפתרון הוא מתן הרשאות חלשות יותר - לכן יש יותר מביט אחד לשליטה במצב (לא המצב ברוב מערכות ההפעלה).

שכבת החומרה מורכבת מפס העברת נתונים יחיד אליו מחוברים כל רכיבי החומרה. פס ההעברה הוא הסטנדרט, מה שאומר שכל רכיב שפועל לפי הסטנדרט, יכול להתחבר, והכל יעבוד כמו שצריך. כיוון שהפס יחיד ומחוברים אליו הרבה רכיבים במקביל, זה משפיע על הביצועים, ומהירות הביצוע תלויה הן בכמות הרכיבים והן באורך הפס. זה נראה, בפשטות כך:



איור 10: פס ההעברה המרכזי אליו מחוברים כל הרכיבים. ניתן לראות שהזכרון מחובר גם לפס וגם למעבד, על מנת לתמוך ב-DMA, כלומר לאפשר לרכיבים פריפריאלים לקרוא מהזכרון מבלי לגשת למעבד. למעשה, לשימוש ב-Bus השלכות רבות על תפקוד המחשב. למשל, אם המעבד סופר מהיר, אבל ה-Bus איטי משמעותית, מה עשינו בזה? כלום. אם המעבד יוצר עומס על הפס, והפס לא מסוגל להעביר את כל המידע, מה עשינו גם כאן? כלום. בהעברת מידע ברשתות תקשורת יש תופעה של פקטות מידע שנעלמות, דבר זה קורה בדיוק כאשר הן מחכות לעבור בפס העברת הנתונים, אך ה-Bus לא מספיק להעביר אותן עקב עומס, והן נזרקות לאבדון.

לכל אחד מרכיבי החומרה יש Controller שהוא מיקרו מעבד בעל קושחה (קוד שמוטבע בתוכו) המסוגל לתקשר עם המעבד בדרך שנקבעת מראש (בעת בניית המעבד).
מערכת ההפעלה מתקשרת עם ה-Drivers, וכיוון שהם רצים במצב הרשאות יחסית גבוה (לרוב Kernel Mode) יש להם גישה לפס ההעברה ולכן הם רושמים בו כתובת שמיועדת לאחד ה-Controllers, כלומר ה-Controller הרלוונטי יזהה את הכתובת, יבין שהכוונה אליו, וכל השאר יבינו שהם צריכים להתעלם (כמו בתקשורת מחשבים בהעברת פקטות).
עתה, הרכיב הרלוונטי ירוץ באמצעות ה-Controller (למשל, אם זה דיסק, הוא יגרום לו להסתובב כדי שיקראו ממנו זכרון), והמעבד יעשה מה שהוא רוצה בינתיים (כאמור, ה-Controller תומך ב-DMA).
עולה השאלה, כיצד יתקשר הרכיב הרץ עם המעבד? כלומר כיצד הוא יגיד לו שהוא סיים לרוץ? בכלל, כיצד הם יעבירו מידע אחד לשני? התשובה היא Interrupts.

2.2 Interrupts

Interrupts הם תנאים מיוחדים שגורמים למעבד לא לבצע את הפקודה הבאה לפי ה-PC, אלא להיכנס ל-Kernel Mode ולהריץ קוד במערכת ההפעלה שמטפל ב-Interrupt שהתקבל.

רכיב פריפריאלי שרוצה להודיע למעבד שהוא סיים לרוץ, שולח סיגנל סיום, שהוא חלק מפקודת Interrupt. יש גם Interrupts מידיים, למשל כל הקלקה על העכבר יוצרת Interrupt, וכל הקלדה על תו במקלדת.

בגדול, רכיבים משתמשים ב-Interrupts כדי להודיע שהם צריכים שירות מהמעבד, דרך מערכת ההפעלה. כלומר, מערכת ההפעלה רואה את ה-Interrupt ומעבירה את המידע הרלוונטי למעבד שיריץ את הקוד המתאים. אממה, כיצד יודע ה-CPU איפה הקוד נמצא?

2.2.1 ה-Interrupt Vector

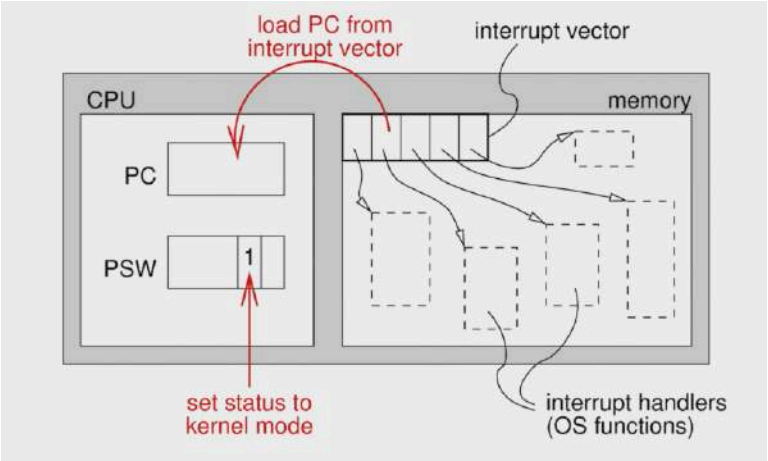
ראשית נבין את הקושי במתן המידע הנ"ל למעבד. ראשית, מדובר בעבודה של המעבד ושל מערכת ההפעלה ביחד - הקוד של מערכת ההפעלה, וההרצה של המעבד.

אממה, המעבדים שרצים על המחשבים שלנו הם לרוב בלתי תלויים במערכת ההפעלה, כלומר, מעבד שיוצר על ידי חברת Intel יכול להיות חלק ממחשב שרצה עליו כל אחת ממערכות ההפעלה Windows/MacOS/Linux.

לכן, מי שצריך לדאוג לספק את המידע הזה, הוא ה-CPU בעצמו, כלומר עליו לספק מנגנון, עליו מערכת ההפעלה תתלבש, וביחד יוכלו לספק לחומרה פוינטרים לפונקציות של מערכת ההפעלה. על כן, החומרה מקצה בלוק בזכרון שמיועד לפוינטרים לקוד ה-Interrupts. הבלוק במקום האפס יכיל כתובת לקוד של ה-interrupt הראשון וכל הלאה, כאשר כל כתובת היא בגודל $4-8$ בתים. $(\text{sizeof}(\text{void}*))$. למעשה, כאשר אנחנו מפעילים את המחשב, אחד הדברים הראשונים שמערכת ההפעלה עושה היא למקם את הכתובות בזכרון של ה-interrupts בבלוק הנ"ל, וזו אחריותה הבלעדית של מערכת ההפעלה. לבלוק הזה בזכרון אנו קוראים interrupt vector.

סך הכל, כאשר controller שולח interrupt, מה שקורה זה שהוא שולח את מספר ה-interrupt לפי מיקומו ב-interrupt vector, והמעבד ילך ל-interrupt vector וישים את הכתובת שנמצאת שם בתוך ה-PC "על עיוור", במחשבה "יש שם 4 בתים, אמור להיות שם קוד חוקי של Interrupt, לכן זה 'אמור' לעבוד". כמובן, מעדכנים את ביט המצב ל-Kernel Mode.

בגדול זה נראה כך:



איור 11: ניתן לראות את ה-interrupt vector המכיל כתובת בזכרון של פונקציות של מערכת ההפעלה. בנוסף, ביט המצב משתנה ל-kernel mode.

הערה. הקריאה של הכתובת מהזכרון על ידי המעבד היא מאוד רגישה, כיוון שאם נוכל לשתול את הקוד במיקום של קודי ה-Interrupts נוכל לאלץ את המחשב להריץ את הקוד שלנו. או, אם ננסה לשנות את הכתובות ב-Interrupt Vector לקוד שלנו, נוכל לעשות דבר דמה. יחד עם זאת, מערכת ההפעלה מגנה על הזכרון **שלה** ולכן מה שיקרה כשננסה לגשת לזכרון הנ"ל, הוא זריקה של Exception.

ת"ז Interrupt כאשר אנו שותלים רכיב חדש בחומרה, אחד הדברים שהמחשב עושה הוא ליצור Interrupt עבורו, עם מזהה מתאים.

שאלה כיצד Controller מודיע למעבד על Interrupt?

תשובה ב-Controll Bus יש Signal Lines (ניתן לראות באיור שצורף קודם לכן), שמאפשרים להעביר הודעה על Interrupt.

הערה. יש מוסכמה ש-$0x80$ interrupt (הפרעה 128) כלומר int 80h הוא ה-interrupt ל-system call.

מסקנה. פקודת ה-Trap היא קריאה int 0x80,
כלומר קריאה למעבד לטעון את הכתובת של ה-switch הענק אל תוך ה-PC מתוך Interrupt Vector [0x80].

2.2.2 Masking Interrupt

מערכת ההפעלה מגיבה ל-Interrupts ואחרת לא עושה הרבה. דבר זה בעייתי - מה אם נשלח interrupt במהלך ביצוע של interrupt אחר? או במהלך הרצה של קוד קריטי? מערכת ההפעלה מסוגלת למנוע קבלת interrupts בדיוק במטרה למנוע אובדן של interrupts שרצים או פגיעה בקוד קריטי שרץ. יש interrupts שלא ניתן לדחות.

כלומר, רק כאשר היא סיימה לטפל ב-Interrupt הנוכחי היא תעבור לטפל באחד הבא. לדחייה וביטול של interrups אנו קוראים masking interrupts.

2.2.3 סיווג Interrupts

נציג סוגים חשובים של interrupts:

1. Interrupts אסינכרוניים חיצוניים (פסיקות חומרה) - נוצרים על ידי רכיבים חיצוניים בזמנים שלא ניתנים לחיזוי מראש. למשל:

(א) Interrupt של השעון - זו דרך של השעון לומר למערכת ההפעלה שפרק זמן מסוים עבר. מי שמטפל בה היא מערכת ההפעלה, ולפעמים היא אף משנה את האפליקציה הנוכחית שרצה. ללא Interrupt זה, האפליקציה עלולה לרוץ לנצח ולהשתלט על המחשב.

(ב) רכיבים פריפריאלים.

(ג) Inetrrupts של כרטיס הרשת - לעיתים מופעל על ידי גורם חיצוני, שיכול להיות גם מחשב מרוחק, שגורם על ידי שליחת הודעה, לכרטיס הרשת לשלוח interrupt למעבד, על מנת שיוכל לעבד את המידע שהמחשב שלח.

2. Interrupts סנכרוניים פנימיים - Exceptions, נוצרות ממערכת ההפעלה, Interrupts של התוכנות שלנו.

2.2.4 הבהרה - trap and interrupt

כאשר אנו מתכנים באסמבלי, אחת הפסיקות המוכרות היא ה-int 21h שמאפשרת, עם פרמטרים שונים, לפתוח קובץ, ליצור קובץ, לקרוא תו מהמשתמש וכו'. עולה מכאן שהיא מכילה גם system calls אבל גם פסיקות חומרה. זה נשמע מוזר, כיוון שאנו יודעים שפסיקות תוכנה אפשר לבצע באמצעות trap 80h וקוד מתאים. ספציפית, זה אומר שיש שתי דרכים לבצע פסיקות תוכנה:

```
; 80x86 assembly code
; using int 21h
mov si, offset outhandle
mov cx, permissions ;attributes actually
mov ah, 3Ch
int 21h

; using trap
mov ax, 8h
mov bx, offset output_name
mov cx, permissions ; attributes actually
int 80h ; can also use "syscall" command instead
```

אם כך מה ההבדל? התשובה היא שמערכות הפעלה שונות בעלות קוד אסמבלי שונה. הפסיקה int 21h היא פסיקת DOS, שזו קבוצה של פסיקות שנכתבה על ידי מייקרוסופט עבור מערכת ההפעלה העתיקה $MS-DOS$. במערכת הפעלה זו, הדרך בה עובדים עם קבצים היא באמצעות אובייקט בשם handle, זו גם הסיבה שכשמשתמים ב-int 21h מקבלים אובייקט handle.

אממה, הדבר קיים רק במערכות ההפעלה של מייקרוסופט, ובמערכות הפעלה שהן Unix Based - שורת קוד זו לא תתקמפל, שכן הקוד לא קיים בכלל. לכן במערכות הפעלה מסוג זה, וב-Linux בפרט, משתמשים ב-int 80h, כלומר ב-trap, או בפקודה syscall שמבצעת אותו דבר.

על כן אנו מבינים מכאן שמערכות ההפעלה של מייקרוסופט, מאפשרות ליצור קובץ בשתי דרכים, אך אחת מהן (DOS−; int 21h) "מותאמת" **יותר** עבור מערכת ההפעלה.

נעיר כי הקוד שהצגנו כאן מתאים ספציפית למערכת ההפעלה $MS-DOS$. אם נרצה לכתוב זאת בלינוקס (כמובן, רק int 80h ולא int 21h, שכן אין שם פסיקות DOS), הסינטקס השתנה, אך הרעיון זהה.

2.2.5 סוגי גרעינים (Kernels)

1. מונוליטי - (old) $MS-DOS$, UNIX. מערכות אלה מכילה קובץ אחד שמכיל את כל הפונקציות שמגיבות ל-Interrupts.

   (א) כיום יש יותר מדי שורות שדרושות עבור Interrupts.

2. Loadable Monotlithic - במטרה לחסוך בזכרון, אפשר תוך כדי ריצה לטעון תכניות שמתמודדות עם Interrupts.

3. Microkernels - גרעין כמה שיותר קטן, ולכן מכיל פעולות בסיסיות ביותר (כמו הפעלה של הרכיבים). אפליקציות במצב משתמש יבצעו את השאר, אך ניהול מערכת הקבצים או טיפול ב-System Calls אינה בהכרח באחריותו, לכן זה מאפשר הרצה של כמה מערכות הפעלה שונות בו זמנית, כי שרתים שונים יכולים לטפל ב-System Calls שונות. זה היה פופולרי בשנות ה-$80-90$.

3 תהליכים וחוטים (תהליכונים) (Processes & Threads)

3.1 תהליכים Processes

הגדרה. תהליך (Process/Job) הוא מופע רץ של תכנית. הוא אובייקט של מערכת ההפעלה. כאשר תכנית מסיימת לרוץ, התהליך "מת". כאשר תכנית רצה, מערכת ההפעלה איננה בתמונה.

שאלה כיצד תכנית הופכת לתהליך?

בגדול, מתבצע ההליך הבא:

כלומר מתבצעים שלושה שלבים:

1. קומפילציה של קובץ הקוד. לפעמים נקמפל לשפת ביניים (VM) שממנה אפשר לתרגם לשפות מכונה של חומרות שונות. נקבל קובץ הרצה.
2. קובץ ה-Executable הוא בפורמט שמתאים לחומרה של המחשב. למשל, a.out לא ירוץ ב-Windows ו-a.exe לא ירוץ על Unix. המידע שנשמר שם, הוא בין השאר, כל המשתנים שהגדרנו ופקודות ההרצה, כדי שמערכת ההפעלה תקצה מקום עבורם. במקרה של הקצאה, לא צריך שכל המקום יהיה בקובץ, למשל, אפשר להקצות מקום עבור משתנים, אבל רק להצהיר על כך.
3. טעינת התכנית לזכרון. קובץ ההרצה נטען לזכרון המחשב (בנאנד קראנו לאזור בזכרון המתאים לה ה-ROM), ויש חלוקה לסגמנטים כמתואר בתרשים, שעליה מפקחת מערכת ההפעלה. בין השאר, נטענים המשתנים של התכנית לזכרון בסגמנט ה-Data.

3.1.1 זכרון תהליך

זכרון התהליך הוא הזכרון הזמין לו במהלך ריצתו, והוא מחלוקת ל-4 סגמנטים:

1. stack - אחראי על קריאות לפונקציה ומשתנים לוקליים (כדי לאפשר קריאות רקורסיבית).
2. heap - הקצאות דינאמיות, למשל באמצעות malloc.
3. data - מכיל את המידע הסטאטי, הגלובלים, אתחולים מ-a.out.
4. text - הפקודות מ-a.out.

הערה. מערכת ההפעלה דואגת שלתהליך לא יהיה את כל הזכרון שהוא רוצה, למרות שהוא מאמין שיש לו.

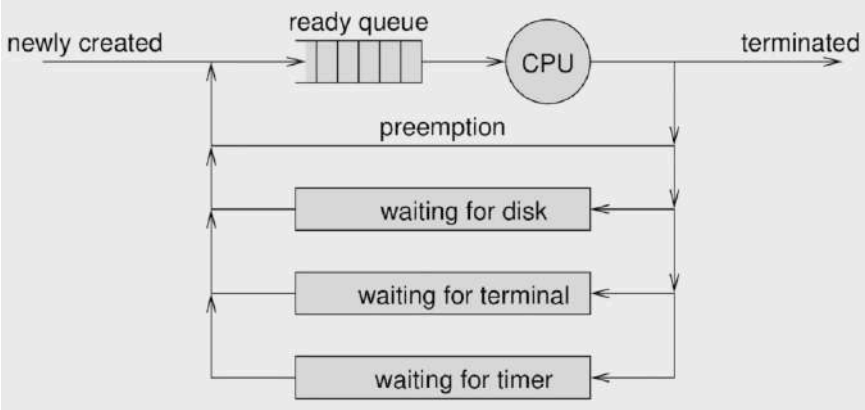
על אף שכל תהליך הוא מופע רץ של תכנית, אין ביניהם התאמה אחד לאחד, שכן ניתן להריץ תכנית בכמה מופעים במקביל, ועל אף שמדובר במופעים של אותה תכנית, מדובר בתהליכים שונים.

3.1.2 מצבי תהליך

תהליך לא רץ כל הזמן במהלך קיומו. קיימים לו שלושה מצבים.

- Running - ממש רץ על ה-CPU
- Ready - מוכן לרוץ, אבל מערכת ההפעלה החליטה להריץ תהליך אחר.
- Blocked - לא יכול לרוץ. מחכה לאירוע מסוים, למשל, לקרוא זכרון מהדיסק, או סיום פעולה של רכיב פריפריאלי.

יחד עם שלושת מצבים אלה, מחזור החיים של תהליך נראה כך:

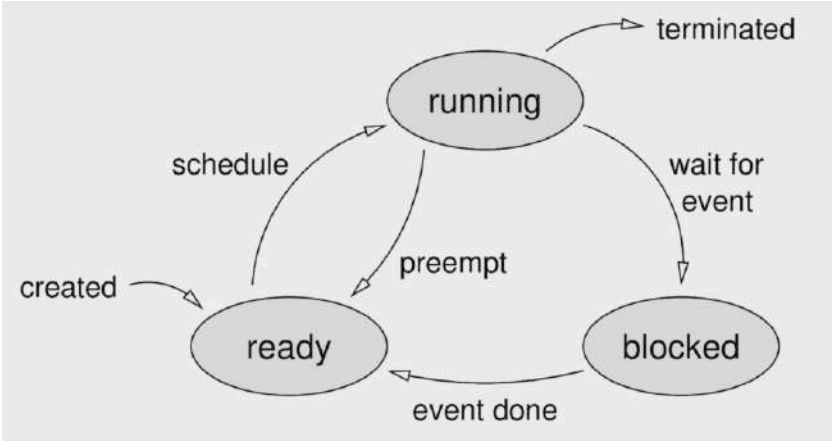


איור 12: תחילה מערכת ההפעלה יוצרת את התהליך, בשלב זה התהליך לא "מיד רץ", שכן מערכת ההפעלה היא זו שרצה. לאחר מכן הוא נכנס לתור התהליכים שמוכנים לרוץ, וכשמגיע תורו הוא עובר דרך ה-CPU, אך על ידי קריאה למערכת ההפעל הוא יכול לבקש זכרון, ולכן יעבור למצב Blocked, וכשיסיים יחזור למצב Ready. לבסוף כשיסיים את כל העיבודים שלו, יעבור למצב terminated.

שאלה באיור הנ"ל, מה הם הרכיבים של מערכת ההפעלה, ושל החומרה?

תשובה ה-CPU הוא רכיב החומרה, והדיסק, טרמינל והטיימר הם של מערכת ההפעלה. זה נכון שבפועל יש שם רכיבי חומרה, אך מערכת ההפעלה נכנסת לפעולה כדי לקשר אותם לתהליך.

איור זה, ניתן גם להציג בצורת דיאגרמה, שמדגישה את "מחזור החיים".



איור 13: הכל מתחיל מיצירת התהליך, לאחר מכן, מערכת ההפעלה, באמצעות Scheduling, והשמה שלו בתור ה-ready, בוחרת מתי להריץ אותו. כשהיא מריצה אותו יש שתי אפשרויות, או שהיא מחליטה לעבור להריץ תהליך אחר ולכן מחזירה אותו למצב ready, או שהוא מחכה לאירוע מסוים ולכן עובר למצב blocked, או שהוא סיים את ריצתו. כאשר הוא במצב ready התהליך חוזר פעם נוספת, וכאשר הוא במצב blocked, הוא מחכה שהאירוע יסיים והוא יוכל לחזור לתור התהליכים המוכנים, הרי זה מצב ready.

3.1.3 יצירת תהליך

אנו יוצרים תהליכים כל הזמן. למשל, כאשר אנחנו נכנסים לחשבון שלנו במחשב, נוצר תהליך שמייצג את המשתמש שלנו, והוא מחכה לבצע פקודות שניתן לו. כיצד?

הוא מתחזק shell או ממשק גרפי לשולחן העבודה, וכאשר אנו מקלידים פקודה ב-shell הוא יוצר תהליך ומריץ אותה, או, אם אנחנו לוחצים פעמיים על אייקון של אפליקציה, מנהל שולחן העבודה יוצר עבורנו תהליך עבור האפליקציה.

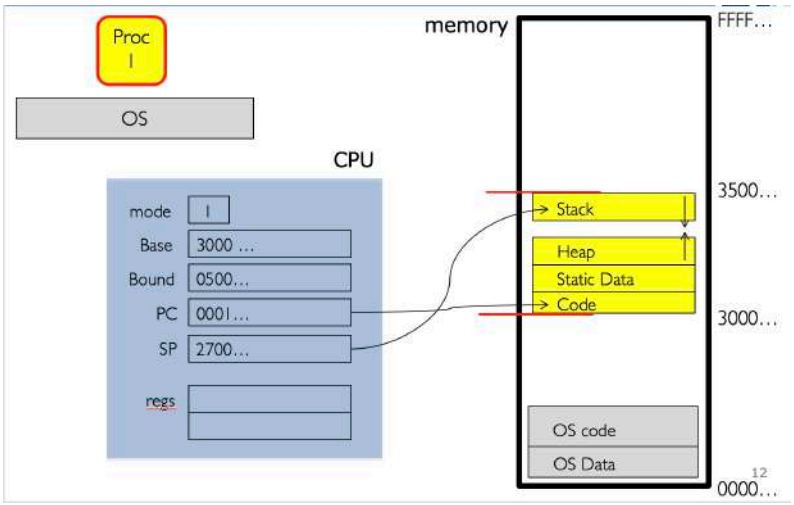
היצירה של התהליכים מתבצעת באמצעות מערכת ההפעלה, על ידי שימוש ב-system call ייעודי. למעשה, כל תהליך יכול ליצור תהליכים נוספים באמצעותן.

שאלה האם צפייה בקובץ היא תהליך?

תשובה קובץ בפני עצמו איננו תהליך, אך ההצגה שלו, כלומר הממשק הגרפי הוא תהליך.

3.1.4 מצב הריצה

מצב הריצה, כאמור, הוא המצב בו התהליך רץ על CPU. מצב זה ניתן להמחשה באיור הבא:



איור 14: ויזואליזציה לריצת התהליך

התהליך מחולק לשני רכיבים - זכרון ומעבד.

מעבד: התהליך רץ ב-User Mode כלומר $\text{Mode} = 1$. הוא מכיל פוינטר SP לסטאק, פוינטר בסיס לפקודות התכנית PC וערך Bound שמסמל חסם על הזכרון עבור זכרון התכנית שאפשר לגשת אליו. כלומר, כאשר המעבד יקבל פקודה או כתובת בזכרון, הוא ישווה את הזכרון שלה ל-Bound, ואם היא עוברת אותו, הוא יזרוק Exception, ככה, על אף שהתהליך חושב שיש לו גישה בלתי מוגבלת לזכרון, הוא מוגבל מאוד. כמו כן, למעבד רג'יסטרים ששומרים את המצב הנוכחי של התהליך במעבד, ולהם יכולת אחסון מוגבלת.

זכרון: כאמור, לתהליך זכרון שמחולק לארבעה סגמנטים. אבל, יש לו אפשרות לשלוח System Calls למערכת ההפעלה, ולכן מוצג זכרון הקוד של המערכת, והמידע שלה.

הקשר הריצה (Context Of Execution) במהלך ריצת תהליך, צריך לזכור את הקשר הריצה, כלומר מה הוא בדיוק עושה ברגע נתון. הקשר זה מחולק לשלושה סגמנטים, והם המעבד, זכרון התהליך, וזכרון מערכת ההפעלה.
המעבד: שומר את ערכי הרג'יסטרים הכלליים וגם את המידע של התכנית. כמו כן, שומר את הרג'יסטרים הספציפיים- PC, SP ורג'יסטרים הקשורים למיפוי זכרון.
זכרון התהליך: התוכן של התהליך במרחב הזכרון המוקצה לו, כלומר ה-data, stack, heap, text.
זכרון מערכת ההפעלה: (מבנה הנתונים המייצג את התהליך) המזהה של התהליך, קבצים שהוא פתח במהלך הריצה, ומידע ניהולי - למשל, אם התהליך רץ לאט יותר או מהר יותר מהתהליכים אחרים, מערכת ההפעלה תזהה את זה ותתזמן אותם בהתאם.

3.1.5 זכרון מערכת ההפעלה (PCB) Process Control Block

מערכת ההפעלה מייצגת תהליך באמצעות מבנה נתונים שנקרא PCB. זהו למעשה Struct עם כל מיני שדות.

המבנה מכיל מידע על הקשר הריצה, שתואר ב-"זכרון מערכת ההפעלה" מקודם. כמו כן, כאשר מערכת ההפעלה מנהלת תהליכים, היא מסתכלת על ה-PCB שלהם, שכן מבחינתה, הם תהליכים לכל דבר. על כן, כאשר היא מייצרת תורים, כמו למשל תור ה-Ready היא יוצרת רשימה מקושרת של PCBs, כאשר המצאות של תהליך ברשימה מסויימת מעידה על מצבו הנוכחי (Ready/Blocked/Running).

השדות של ה-PCB, בגרסתו המופשטת הם:

1. מזהה התהליך - (ID)
2. מצב התהליך - (Ready/Blocked/Running) עם פירוט, למשל אם הוא Blocked אז למה הוא מחכה.
3. משתמש - מספק מידע על ההרשאות של התהליך, למשל אם הוא מבקש לגשת לזכרון מסוים, יתכן שאין לו בכלל הרשאה.
4. משאבים שהתהליך צרך -
   (א) מהירות הריצה.
   (ב) עדיפות התהליך.
5. מצביעים לזכרון -

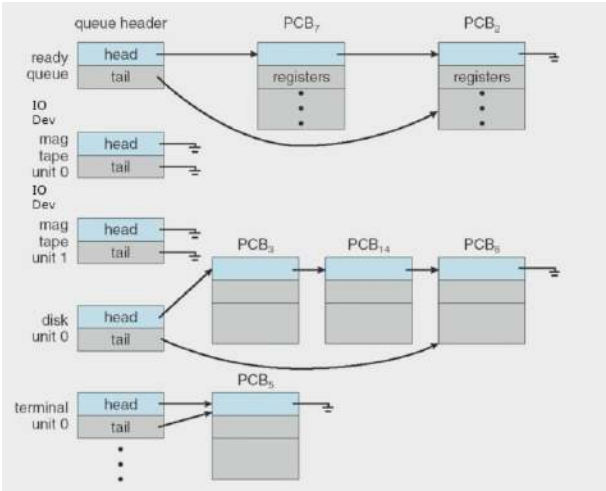
(א) זכרון מוקצה.

(ב) קבצים פתוחים.

(ג) ערוצי תקשורת.

6. מקום לשמור את מצב ה-CPU - שכן, כאשר התהליך הוא לא במצב "ריצה", צריך לשמור על המצב שלו במעבד כדי שיוכל להמשיך לרוץ מהנקודה שבה עצר. כשהוא רץ המקום מבוזבז.

נמחיש אם כך את תורי מערכת ההפעלה המורכבים מרשימות מקושרות של PCBs:



איור 15: כל רשימה מקושרת מורכבת מ-PCBs, כאשר לכל רשימה גישה לראש ולזנב.האיור מאוד ישן, שכן מוצגים בו רשימות של טייפ מגנטי.

3.1.6 סיום תהליך (Process Termination)

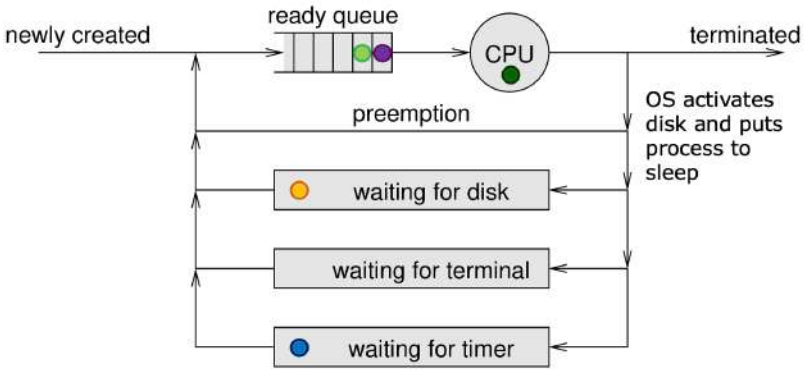
תהליך יסיים את ריצתו עקב הרבה מאוד סיבות. ביניהן:

1. סיום ריצה תקנית (מרצונו החופשי של התהליך) - הכנית סיימה את ריצתה.
2. סיום ריצה עקב שגיאה (מרצונו החופשי של התהליך) - למשל הרצנו קומפיילר כדי לקמפל קובץ foo.c שבכלל לא קיים. התהליך ירוץ, אבל יחזיר הודעת שגיאה ויעצור.
3. שגיאה קטלנית (לא מרצונו החופשי) - למשל חילקנו באפס. יישלח Exception על ידי המעבד, ואם הוא לא ייתפס, מערכת ההפעלה תהרוג את התהליך.
4. הריגה על ידי תהליך אחר (לא מרצונו החופשי) - למשל, הרצנו תכנית דרך הטרמינל והחלטנו שאנחנו רוצים לעצור אותה. אנחנו לוחצים על ctrl c, ה-shell קולט את זה ושולח סיגנל למערכת ההפעלה על כך שהוא רוצה להרוג תהליך, באמצעות system call, כלומר, הוא רואה את התהליך הנוכחי שרץ, שכן הוא הריץ אותו, ושולח אליו סיגנל במטרה שמערכת ההפעלה תהרוג אותו. היינו רוצים שלא כל תהליך יוכל להרוג תהליכים אחרים, ולכן זה תלוי בהרשאות שלהם. במקרה הנ"ל, ל-shell יש הרשאות, בפרט הוא זה שיצר את התהליך.

3.2 ריבוי תהליכים (Multiprocessing)

מערכת ההפעלה מכילה מספר תהליכים הרצים במקביל, ולכן כפי שתיארנו, היא דואגת להפרדה בין התהליכים על ידי מתן זכרון מוגבל לכל אחד.

מעבר לכך, היא צריכה להיות מסוגלת לתזמן את התהליכים השונים לפי המידע שיש לה. נמחיש תהליך זה באמצעות האיור הבא:

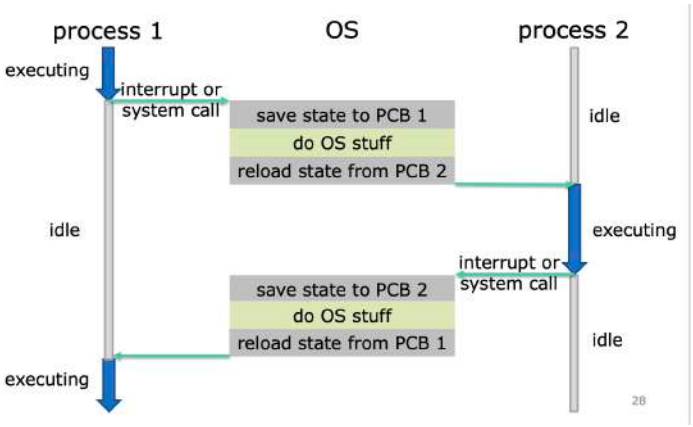


איור 16: תחילה התהליך נוצר ורץ ובמהלך הרץ הוא ביצע את הפקודה fork שיצרה תהליך חדש (הירוק בהיר). התהליך החדש נכנס לתור, אך מערכת ההפעלה החליטה שיש לו עדיפות גבוהה יותר ולכן החליטה להריץ אותו, לכן היא מכניסה את התהליך הירוק כהה לתור מחדש ומריצה את הירוק בהיר. לפתע מגיע interrupt מהשעון ומערכת ההפעלה מריצה את הקוד שמטפל בו, וכשהיא מסיימת היא נזכרת שיש תהליך כחול שמחכה לשעון ולכן עליה לשחרר אותו, ולהכניס אותו לתור ready. היא משחזרת את המידע של התהליך הירוק להמשך הרצה, אך מגלה שלתהליך הכחול עדיפות גבוהה יותר ולכן מריצה דווקא אותו ומכניסה את התהליך הירוק בהיר לתור ה-ready.

3.2.1 Context Switch

במהלך הכנסת כל תהליך לתור והרצת תהליך אחר, מערכת ההפעלה מבצעת פעולת Context Switch, כלומר מחליפה את הקשר הריצה עליו דיברנו קודם. מה קורה בפועל?

1. נשמור את הקשר התהליך הנוכחי ב-PCB שלו, כולל הרג'יסטרים וכל מה שתיארנו קודם.
2. נעדכן את ה-PCB של התהליך הנוכחי והתהליך החדש במצב החדש שלהם (למשל, תהליך Blocked הפך ל-Running)
3. נעביר את ה-PCB לתור הרלוונטי, למשל תהליך שהיה Running הפך ל-Ready/Blocked.
4. נשחזר את הקשר הריצה מהתהליך החדש אל תוך ה-CPU, ורק אז נוכל להתחיל להריץ את התהליך החדש.

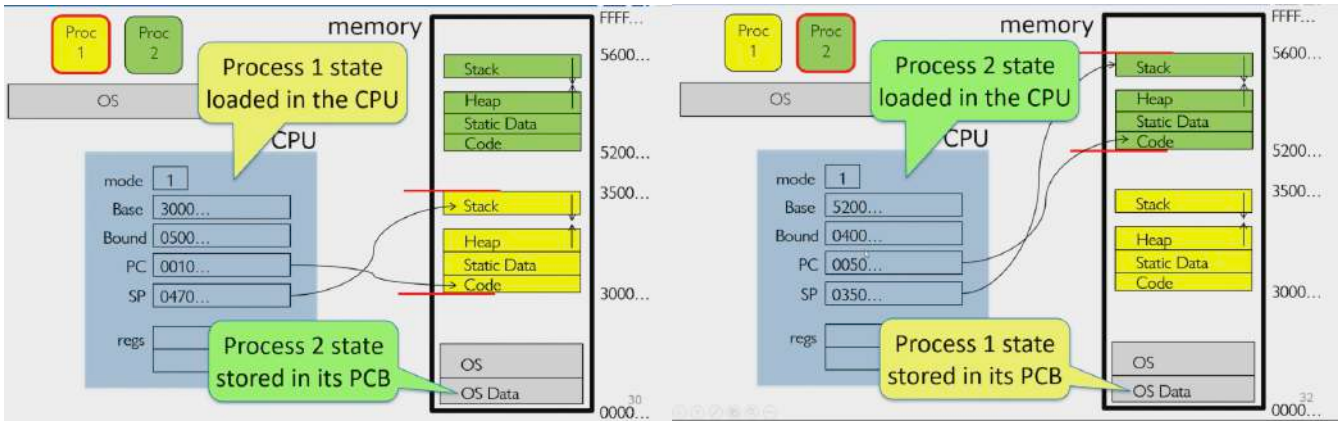


איור 17: תחילה התהליך הראשון רץ ופרק זמן הריצה מסומן בחץ כחול. לאחר מכן מתבצעת פסיקה בשביל System Call שגורמת למערכת ההפעלה לשמור את הקשר הריצה ל-PCB_1, לבצע פעולות ביניים ולטעון את הקשר הריצה של תהליך חדש מ-PCB_2. עתה, רץ התהליך השני, ושוב יש פסיקה וקריאה למערכת ההפעלה. נשמר הקשר הריצה של התהליך השני, מערכת ההפעלה מבצעת שוב פעולות ביניים, וטוענת מחדש את הקשר הריצה של תהליך 1 מתוך PCB_1, והתהליך הראשון ממשיך לרוץ מהנקודה שבה עצר.
כל הזמן עד להתחלת הריצה של התהליך השני/הראשון הוא Overhead, אותו אנו רוצים למזער. אנו מעוניינים בתהליכים שרצים.

שאלה מה בדיוק צריך לשמור כאשר מבצעים Context Switch?

תשובה באופן כללי, מעבד וזכרון. אבל הזכרון כבר "נמצא בזכרון" ולכן אין צורך להעתיק אותו. המעבד לעומת זאת יחיד ולכן יש צורך להעתיק את כל המידע שבו, כולל SP, PC ועוד.

בפועל, זה נראה כך:



איור 18: באירו השמאלי ניתן לראות את מצב המעבד והזכרון במהלך ריצתם של שני תהליכים. תחילה המעבד מכיל את הקשר הריצה של תהליך 1, והזכרון מכיל את הזכרון שלו בצהוב ואת הזכרון של תהליך 2 בירוק, ואת מערכת ההפעלה והמידע שלה, המכיל את ה-PCB_2.
באיור הימני, רואים מצב בו לפתע מתבצע Context Switch והתהליך שרץ עכשיו הוא תהליך 2, וניתן לראות את ההחלפה של הקשר הריצה על ה-CPU ל-PCB_2. הזכרון לעומת זאת לא השתנה, מלבד התוכן של PCB_1, PCB_2, שכן מצב התהליכים השתנה, והם שמורים במידע של מערכת ההפעלה, והמידע של התהליכים, נשמר בזכרון באותו מקום. רק מה שמשתנה הוא "קטע הזכרון הרלוונטי" עבור המעבד, שכן PC, Bound

שאלה מתי אנחנו מבצעים את ה-Switch?

תשובה כאשר יש פסיקות, כמו שעון או רכיבים פריפירלאים. אם יש שגיאת זכרון, קריאה למערכת ההפעלה שדורשת המתנה, חריגה שגורמת להרג התהליך.

שאלה מי מחליט את מי להריץ ומי מבצע?

תשובה שני גורמים: Dispatcher, CPU Scheduler.

CPU Scheduler רץ על המעבד, מתזמן לטווח הקצר ומבצע זאת מאוד פעמים בשנייה.

Dispatcher מי שאחראי על ביצוע ההחלטות שנקבעות על ידי ה-CPU Scheduler והם Context Switch וחזרה למצב משתמש ממצב גרעין.

3.2.2 inter − process Communication (IPC)

כאשר תהליכים רצים, יש אפשרות לבצע תקשורת בין התהליכים, על ידי יצירת System Call ייעודי. דבר זה בעל Overhead גבוה, שכן התקשורת עוברת תמיד דרך מערכת ההפעלה. בין השאר אפשר ליצור:

- זכרון משותף לתהליכים (Shared Folder), שכל התהליכים יוכלו לגשת אליו, כמובן זה דורש ניהול מיוחד על ידי מערכת ההפעלה.
- העברת הודעות באמצעות ערוץ תקשורת (Socket).
- קריאת וכתיבה לקבצים מתהליכים שונים.

3.3 חוטים (תהליכונים) (Threads)

Thread הוא נתיב ביצוע של תהליך. עד כה התייחסנו לתהליך כאילו הוא מריץ את הפקודות אחת אחרי השנייה עד שהוא מסיים, כאילו שיש לו Thread יחיד שהוגדר על ידי ה-PC, Stack והתחיל מה-main.

אבל בפועל, אפשר להריץ חלקים שונים, בלתי תלויים של הקוד במקביל באמצעות Threads.

כל נתיב יבצע חלקים שונים של אותו הקוד, במקביל, כאשר יש להם הקשר משותף הכולל את הזכרון והקבצים הפתוחים. אך לא את המעבד, שכן כל אחד מהם רץ במקום אחר.

למשל, חוט אחד יכול לקרוא מקובץ וחוט שני לכתוב אליו.

3.3.1 ריבוי תהליכים (Multi − Threaded)

שאלה מתי צריך Threads?

תשובה הדפדפן שלנו, אנחנו רוצים שהוא יהיה זמין אלינו כל הזמן, אבל יקרא מידע מהרשת בו זמנית. עורכי טקסט, הוא צריך גם לקלוט מידע מאיתנו אבל גם להציג את המידע על המסך. שרת אינטרנטי, צריך להיות זמין לעבד בקשות חדשות, ולעבד בקשות ישנות במקביל.

שאלה כיצד עושים את זה?

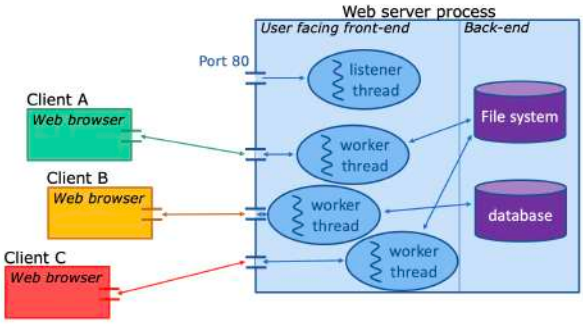
אפשרות אחת היא על ידי שימוש ב-Thread אחד, כלומר הרצה של פקודה אחת בכל פעם. הבעיה היא שזה איטי מדי שכן אנחנו הרצה במקביל.

אפשרות נוספת היא הרבה IPC, כלומר הרצת תהליכים מרובים ותקשורת דרך מערכת ההפעלה. זה אמנם יעבוד, אבל יהיה איטי מאוד, שכן יש לזה הרבה מאוד Overhead.

לכן נציע שימוש בתהליכים $\text{Multi} - \text{Threaded}$, כלומר בעלי ריבוי חוטים. בתהליך כזה, כל משימה בעלת חוט משלה, ובעלת הקשר ריצה משלה $(\text{PC}, \text{SP}, \text{Stack}, \ldots)$ וכולם משתפים את קוד התכנית, משתני התכנית, זכרון, קבצים פתוחים וערוצי תקשורת. היתרון הוא שאין כאן צורך במערכת ההפעלה לתקשורת, שכן יש שיתוף.

הבדל בין חוטים ותהליכים לחוטים ותהליכים יש הבדל מהותי. תהליך יוצר וירטואליזציה של המחשב, והוא חושב שכל המחשב מיועד בשבילו. חוט לעומת זאת, מייצר וירטואליזציה של המעבד. ניתן לבצע הרצה אמיתית במקביל על ידי הרצה של חוטים שונים על מעבדים שונים, או הרצה כאילו "במקביל" על ידי הרצה על אותו מעבד, עם Time Slicing.

דוגמה. נציג המחשה לשרת Web בעל ריבוי חוטים. ויזואלית, זה יראה כך:



איור 19: השרת מכיל חוט אחד שאחראי על משימת התקשוב - ללקוחות שרוצים להתחבר.
נניח שהחוט הנ"ל הקשיב ללוקח A. הוא ייצור Thread חדש במיוחד עבורו שיהיה אחראי על התקשורת של הלוקח עם מערכת הקבצים ומסד הנתונים.
לפתע הוא מקשיב גם ללקוחות B, C ולכן הוא מקצה עבורם Thread ייעודי באופן דומה, בעצם, כל אחד מהם $(\text{Thread} - \text{A}, \text{B}, \text{C})$ הוא עותק של אותה משימה. יצירת ה-Thread מתבצעת על ידי System Call ייעודי, כלומר אל מול מערכת ההפעלה.
כמו כן, ניהול נכון של החוטים יוצר עבודה טובה במקביל ושרת מוצלח.

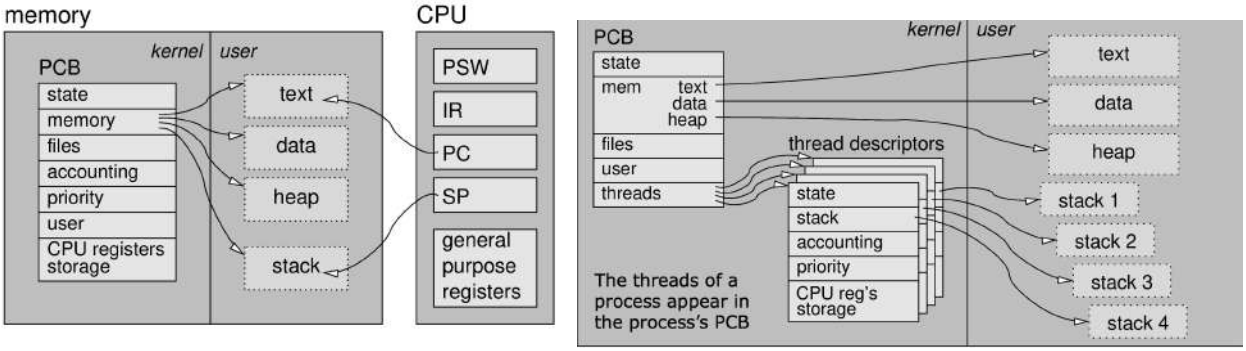
נעיר כי במקרה זה אין ממש שיתוף זכרון. במקרה של עורך טקסט יש שיתוף נרחב יותר, שכן יש גישה של כולם לקובץ המכיל את המידע.

עולה השאלה, איפה מערכת ההפעלה? מערכת ההפעלה היא זו שיוצרת את ה-Threads השונים, מחליפה ביניהם, אחראית לניהול חיצוני של התהליך, קריאה מהדיסק, תקשורת (גישה לכרטיס הרשת). כלומר היא נמצאת ברקע של כל התהליך.

3.3.2 סוגי חוטים

יש סוגים שונים של חוטים, שכל אחד מיועד למשימות שונות.

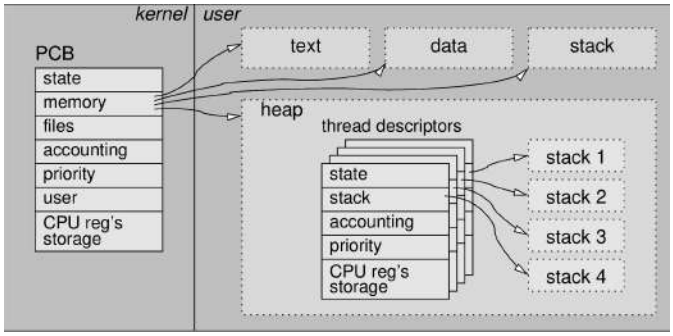
$\text{Kernel} - \text{Level Threads}$ הקרנל מתחזק את החוטים, והם קיימים עבור תהליך משתמש. כלומר, אם חוט רוצה לגשת למערכת הקבצים, הוא היחיד שצריך לחכות, והתהליך שמכיל אותו לא נעצר.
החסרון הוא שמעבר בין Threads בעל Overhead. בהשוואה לתהליכים, נקבל את הדבר הבא



איור 20: בעוד תהליך רגיל מכיל מחסנית אחד, הקשר ריצה אחד וכל מה שתיארנו קודם
תהליך בעל ריבוי חוטים (Kernel − Level) מכיל מחסנית משלו, מצב משלו, הקשר מעבד משלו, עדיפות ועוד, כל זה על מנת שיוכל להריץ פונקציות בעצמו.

Kernel Threads חוטים שמנוהלים על ידי הקרנל ורצים במרחב הקרנל. קיימים עבור מערכת ההפעלה. למשל טיפול בפסיקה - נדרש חוט ייעודי, על מנת לאפשר הרצה במקביל של דברים נוספים.

User − Level Threads כל ניהול החוטים מתבצע על ידי המשתמש ולא על ידי הקרנל, כלומר, נשתמש בספריה כתובה של חוטים, שתספק אובייקט מתאים, ואנו ננהל אותו.
אין כאן באמת הרצה במקביל אלא רק "סימולציה" להרצה במקביל. לכן, הקרנל לא מודע לחוטים, וכאשר הוא מקבל System Call הוא עוצר את כל התהליך, ולא רק את החוט. כמו כן, החלפה בין חוטים לא דורשת את הקרנל. משתמשים בזה כאשר הרצת התכנית מתאימה יותר להרצת תכניות במקביל, אך המחשב לא תומך בהרצה אמיתית במקביל. בנוסף, סנכרון לא נכון של החוטים עלול להביא למצב שכל החוטים לא יקבלו גישה למשאבים הדרושים והתהליך יקרוס.
סך הכל, זה יראה כך:



איור 21: הקרנל (מערכת ההפעלה), לא מודעת לחוטים של התהליך, אלא זו רק וירטואליזציה פנימית של התהליך לחוטים. אנחנו כן יכולים לקרוא לפונקציות מתוך התהליך, ולכן לכל חוט יש "כאילו" מחסנית משלו. (נעשה זאת בתרגיל 2)

3.3.3 יתרונות של חוטים אל מול תהליכים

נשווה בין חוטים לבין תהליכים. לא כאובייקטים הניתנים להחלפה בהכרח, אלא רק כדי לתת סדר גודל.

- יצירת חוט מהירה פי $100 - 30$ מיצירת תהליך.
- סיום ריצה של חוט מהירה יותר מסיום ריצה של תהליך.
- ביצוע Context Switch בין חוטים מהיר פי ~ 5 בהשוואה לתהליכים.
- חוטים בתוך אותו תהליך חולקים את הזכרון והקבצים, ולכן יכולים לתקשר ביניהם מבלי לערב את הקרנל.

נוכל לסכם זאת בטבלה הבאה:

User − level Threads	Kernel − level threads	תהליכים
זהה לחוטי קרנל	חולקים מרחב כתובות, יכולים לבצע תקשורת בסיסית, שימושיים עבור בניית אפליקציה	מוגנים אחד מהשני, יכולים לתקשר אחד עם השני באמצעות מערכת ההפעלה
תקורה נמוכה : הכל מבוצע ברמת המשתמש	תקורה בינונית : פעולות דורשות את הקרנל אבל עבודתו מועטה	בעלי תקורה גבוהה : כל הפעולות דורשות שימוש בקריאות OS
אם חוט נעצר, הוא עוצר את כל התהליך	כמו תהליכים	אי תלות : אם אחד נעצר, זה לא משפיע על האחרים
חולקים את אותו מעבד, ורק אחד רץ כל פעם	כמו תהליכים	יכולים לרוץ במקביל על מעבדים שונים
אותה ספריית חוטים תוכל לעבוד על כמה מערכות הפעלה	כמו תהליכים	כדי לייצר תהליך חדש צריך להשתמש בממשק של מערכת ההפעלה הספציפית. תכניות תלויות במערכת ההפעלה

טבלה 1 : השוואה בין תהליכים לבין חוטים

4 סנכרון (Synchronization)

4.1 מבוא

כשדיברנו על תהליכים ועל חוטים, ראינו שיש להם אפשרות לתחזק זכרון משותף. עבור חוטים - כל הזכרון של התהליך, ועבור תהליכים, זכרון שמוקצה לכך מראש.

נראה כמה בעיות שיכולות לצוץ עקב כך.

ניזכר ב-Spooler, האחראי על ההדפסות למסך במחשב. נניח כי הוא ממש על ידי מערך של פוינטרים, כאשר כל פוינטר מצביע להודעה בזכרון שצריך להדפיס.

על כן, הוא ישמור משתנה IN שיכיל את האינדקס של המקום הפנוי הראשון בתור, ו-OUT, האינדקס של ההודעה שצריך להדפיס. על כן, כאשר תהליך רוצה לבצע ההדפסה, הוא מכניס את הכתובת למקום הפנוי ב-Spooler ומעלה את ערכו של IN.

אם כך, מצב תקין הוא מצב מהצורה:

```
Process 1: Spooler[IN] = JOB1
Process 1: IN++

<Context Switch by OS>

Process 2: Spooler[IN] = JOB2
Process 2: IN++
```

שכן, כל תהליך סיים את מה שהיה צריך לעשות עם ה-Spooler ללא הפרעה של מערכת ההפעלה.

אבל, לתהליכים אין שליטה על ביצוע Context Switch, ומערכת ההפעלה עלולה לבצע אותו במקומות לא אידאלים. למשל, יתכן המצב הבא:

```
Process 1: Spooler[IN] = JOB1

<Context Switch by OS>

Process 2: Spooler[IN] = JOB2

<Context Switch by OS>

Process 1: IN++

<Context Switch by OS>

Process 2: IN++
```

כאן, מערכת ההפעלה עצרה את הגדלת IN על ידי Process_1, וגרמה ל-Process_2 לדרוס את שעשה הראשון. לאחר מכן, מערכת ההפעלה עצרה גם את ההליך השני, והחזירה את השליטה להליך הראשון. ההליך הראשון הגדיל את IN, ולאחר מכן מערכת ההפעלה החזירה את השליטה השני, שהעלה גם את IN,

והתוצאה של כל זה הוא שדילגנו על מקום פנוי בתור, בגלל אי ההגדלה ההתחלתית של IN על ידי התהליך הראשון. כלומר, נוצר רווח בתור, וגם ההליך השני דרס את הקלט להדפסה של הראשון.

זו בעיה מאוד בסיסית, אבל יש עוד הרבה בעיות. למשל, בשתי הדוגמאות הקודמות הנחנו שלא ניתן לעצור את הפקודה $IN++$ באמצע. אבל בפועל, לא מדובר בפקודה אחת שהמעבד מריץ. אלא למעשה, מדובר בשלוש פקודות שהמעבד צריך להריץ:

```
lw r1, 100
inc r1
sw 100, r1
```

כאשר חוטים רבים מבצעים פעולות שונות במקביל, או ב-Time Sharing תתכן ההתנהגות הבאה:

```
Thread 1: lw r1,100
Thread 1: inc r1

<Context Switch>

Thread 2: lw r1, 100
Thread 2: inc r1
Thread 2: sw 100, r1

<Context Switch>

 Thread 1: sw 100,r1
```

תחילה נטען הערך ממקום 100 בזכרון ל-r_1, והחוט העלה את ערכו. אך לפתע התבצע החלפת הקשר ריצה, והחוט השני טען שוב את הערך ל-r_1, העלה את ערכו, וטען חזרה את הערך למקום 100 בזכרון.

עתה, התבצעה החלפת הקשר לחוט 1 והוא טען את r_1 (שנשמר מבעוד מועד) לכתובת 100, ולמרות שרצינו לבצע פעמיים $++$ IN קיבלנו רק פעם אחת.

זו שגיאה שיכולה גם לקרות באופן נדיר, וגם אם נריץ את התכנית עשרות רבות של פעמים, יתכן שהיא קיימת.

הערה. כיצד ניתן לפתור את הבעיה? נוכל להקביל את הבעיה לחיים. נניח שאנו פותחים את המקרר ונגמר החלב, ראינו בקבוק ריק, זרקנו אותו, והלכנו לקנות חלב. אדם נוסף שגר בדירה, רואה מקרר ריק, ולכן הולך גם לקנות חלב. זו דוגמא קלאסית לאנשים בלתי מתואמים. נוכל לפתור זאת על ידי השמת פתק על המקרר ובו נכתוב שהלכנו לקנות חלב. אממה האם בזאת נפתרה הבעיה? לא. אם לא נוריד את הפתק, נתקל בחוסר בחלב בעתיד. לכן עלינו לדאוג גם לזה.

בעיה. גישה למשאב משותף, שאיננה מתאומת.

הגדרה. קטע הקוד הקריטי (Critical Code Section) בגישה למשאב משותף זה קטע הקוד שניגש למשאב. הוא יכול להיות פקודה אחת, או קטע קוד נכבד.

מטרה קטע הקוד הקריטי יבוצע בקטע זמן נתון על ידי חוט אחד בלבד ותהליך אחד בלבד.

הגדרה. Mutual Exclusion. אלגוריתמים שימנעו גישה סימולטנית לקטעי קוד קריטיים, ונכתבים על ידי המשתמש.

אם כך, הפתרון שנציע לבעיה הוא שימוש ב-Mutual Exclusion Algorithms, ונעיר כי לעיתים זה לא מספיק, שכן סדר הביצוע עלול להיות קריטי.

4.2 Mutual Exclusion Algorithms

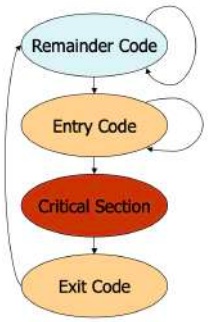
מידול האלגוריתמים הנ"ל התבצע על ידי דיקסטרה (כן, מר מרחק מינימלי בגרף ממושקל).

הוא הציע את המודל הבא.

בהנתן קטע קוד, אנחנו מגדירים את קטע הקוד הקריטי. כל שאר קטע הקוד, נקרא שארית הקוד (Remainder). בין השארית לקריטי, נכניס קוד ביניים שנקרא Entry Code שמטרתו לוודא שהכניסה לקטע הקוד הקריטי היא חוקית, ולא תיצור התנהגות שאנו רוצים להימנע ממנה, כמו שראינו בדוגמאות הקודמות. לכן לרוב מכיל הרבה לולאות שעוצרת את התהליך/חוט,

לאחר הכניסה לקטע הקוד הקריטי, נכנסים לקטע קוד נוסף שנקרא קוד היציאה (Exit Code) וממנו חוזרים חזרה לשארית.

ויזואלית זה נראה כך:



איור 22: המחשה למודל. מטרתנו כמפתחים היא לכתוב את קטעי הקוד של ה-Entry&Exit. כיוון שאנחנו, המשתמשים, כותבים קטע קוד זה, הוא רץ במצב User Mode, ולכן אנחנו לא יכולים "לעשות קסמים" של מערכת ההפעלה.

מימוש המודל צריך לענות על 5 תנאים קריטיים.

1. Mutual Exculsion: אין שני תהליכים שניגשים לקטע הקוד הקריטי באותו הזמן.
   (א) לבד זה לא תנאי מספיק, שכן יתכן שתהליכים לא יבצעו קטע קוד קריטי לעולם.
2. Progress: אם תהליך מנסה לגשת לקטע הקוד הקריטי ולא מצליח, אז תהליך אחר מבצע אותו.
   (א) לבד זה לא תנאי מספיק, שכן יתכן שאנו בעדיפות נמוכה, ותהליכים יעקפו אותנו כל הזמן, ואנו לעולם לא נבצע את קטע הקוד הקריטי.
3. Starvation Freedom: אם תהליך רוצה לגשת לקטע קוד קריטי, אז התהליך בסוף גם יבצע אותו.
   (א) זה תנאי חזק יותר מתנאי 2, וכביכול, מספק את כל מה שרצינו.
   (ב) אין כאן הבטחה על מתי יתבצע קטע הקוד, יתכן שנחכה הרבה זמן.
4. Generality: לא ניתן להניח הנחות על מהירות או מספר המשתתפים (חוטים/תהליכים/ליבות/מעבדים).
   (א) נבטיח את כלליות האלגוריתם. שיעבוד טוב תמיד.
5. No blocking in the remainder: לא יתכן שאנו לא מעוניינים בגישה לקטע קוד קריטי, ואיננו מבצעים אותו, אבל מנענו מתהליכים אחרים לגשת אליו.
   (א) אל לנו לשכוח, יתכן שביצענו קטע קוד קריטי וחזרנו לקוד השארית, אבל עדיין מנענו מתהליך לגשת לקטע הקוד. עלינו לוודא שזה לא יקרה.
   (ב) **זו נקודה מאוד חשובה**.

מסקנה. קטע קוד קריטי מתבצע בזמן סופי, שכן אחרת תנאי 3 לא יתקיים. אנו מניחים שהוא חייב להתבצע בזמן סופי, שכן אחרת המודל לא ישים.

4.2.1 דוגמאות לאלגוריתמי Mutual Exclusion

אם, כך נרצה להציע אלגוריתם שיענה על תנאי המודל.

דוגמה. נציע את האלגוריתם הבא:

- Entry Code: לא נאפשר לבצע Interrupt.
- Exit Code: נאפשר לבצע Interrupts.

על פניו, תנאים $1, 2, 5$ מתקיימים. אבל יש בעיה. המשתמש לא יכול לשלוט בקבלה ודחיית פסיקות, זו אחריות של ה-Kernel.

כמו כן, הוא לא פותר כמה מקרים שיתכנו. למשל, עם כמה ליבות קיימות, הוא מתעלם מהמקרה שקיימת הרצה במקביל אמיתית. כמו כן, דחיית פסיקות צריך להתבצע לזמן קצר מאוד, שכן הוא משהה תקשורת בין החומרה למערכת ההפעלה.

נסיק כי הפתרון לא טוב.

דוגמה. נציע את האלגוריתם הבא:

- נשמור משתנה גלובלי Turn שישמור את המזהה של ה-Thread שיכול לגשת אליו.
- Entry Code: $\text{While}\,(\text{Turn}! = \text{thread}_{\text{id}})\,;$, כלומר כל עוד זה לא המזהה של החוט הנוכחי, חכה.
- Exit Code: $\text{Turn} = \text{thread}_{\text{id}}$.

התנאי הראשון מתקיים, שכן שני חוטים לא יגשו במקביל לקטע הקוד קריטי, כי המשתנה Turn יכול להיות מספר אחד בכל פעם.

האם תנאי התהליך יתקיים? לא. למשל, אם חוט אחד ניגש לקטע הקוד הקריטי, הוא לא עדכן בחזרה את Turn להיות מספר השונה מהמזהה שלו, ולכן כל חוט אחר לא יוכל לגשת לקטע הקוד הקריטי. מכאן גם תנאי 3 לא מתקיים, יש הרעבה.

דוגמה. נציע הרחבה של האלגוריתם הקודם:

- נשמור מערך flag שבמיקום ה-i יכיל True אם החוט ה-i יכול לגשת לקטע הקוד הקריטי ו-0 אחרת.
- Entry Code:

$$\begin{aligned}&\text{flag}\,[\text{Thread}_{\text{id}}] = \text{True};\\&\text{While}\,(\exists \text{i} \neq \text{Thread}_{\text{id}}\ \text{s.t flag}\,[\text{i}])\,;\end{aligned}$$

, כלומר כל עוד זה לא המזהה של החוט הנוכחי, חכה.

- Exit Code: $\text{flag}[\text{Thread}_{\text{id}}] = \text{False}$.

התנאי הראשון מתקיים, שכן שני חוטים לא יגשו במקביל לקטע הקוד קריטי, מהתנאי בלולאת ה-While.

האם התנאי השני מתקיים? נניח שבמהלך ה-Entry Code התבצעה הפקודה הראשונה כלומר הדגל מאפשר הרצה, אך לפתע התבצע גם Context Switch ועברנו לחוט אחר. גם החוט החדש רוצה להריץ את קטע הקוד הקריטי וגם הוא הגדיר דגל בהתאם. אבל הדגל הראשון עדיין דולק, ולכן הלולאה תמשיך לרוץ והוא לא יגש לקטע הקוד לעולם, וכמו כן, בחזרה לחוט הראשון, הדגל של החוט השני דולק, ולכן גם הוא לא יגש לקטע הקוד לעולם.

למצב זה אנו קוראים Dead Lock מצב בו אף אחד לא ניגש למשאב, עקב טעות בסנכרון. לכן, גם התנאי השני וגם השלישי לא מתקיימים.

דוגמה. נציע שילוב של שני האלגוריתמים:

- נשמור מערך flag שבמיקום ה-i יכיל True אם החוט ה-i יכול לגשת לקטע הקוד הקריטי ו-0 אחרת.
- נשמור משתנה עזר Turn
- Entry Code:

$$\begin{aligned}&\text{Turn} \neq \text{Thread}_{\text{id}};\\&\text{flag}[\text{Thread}_{\text{id}}] = \text{True};\\&\text{While}\left((\text{Turn} \neq \text{Thread}_{\text{id}}) \ \text{and} \ (\exists \text{i} \neq \text{Thread}_{\text{id}} \text{ s.t flag}[\text{Thread}_{\text{id}}])\right);\end{aligned}$$

, כלומר כל עוד זה לא המזהה של החוט הנוכחי, חכה.

- Exit Code: $\text{flag}[\text{Thread}_{\text{id}}] = \text{False}$.

האם התנאי הראשון מתקיים? לא, שכן יתכן מצב שהחוט הראשון מכיל אמת במקום שלו במערך הדגלים, אבל זה לא תורו.

למשל, אם ביצענו $\text{Turn} = 1$ מחוט 0, ומיד עברנו לחוט 1. הפכנו $\text{Turn} = 0$ והגדרנו $\text{flag}[1] = \text{True}$. בשלב זה $\text{flag}[0] = \text{False}$ ולכן חוט 1 יכנס לקטע הקוד הקריטי.

עתה, חוט 0 יעדכן את הדגל שלו להיות True, אבל $\text{Turn} = 0$ ולכן הוא יכנס גם לקטע הקוד הקריטי. על כן, התנאי הראשון לא מתקיים.

מסתבר, שעל ידי שינוי קטן בקוד, אפשר לקבל קוד שעובד והוא האלגוריתם הראשון שעבד. כדי לרשום זאת עבור שני חוטים, החוט הנוכחי יסומן ב-i והחוט האחר ב-$1 - i$. מזה נקבל את קטע הקוד הבא, שהוצע על ידי פיטרסון ב-1981

- Entry Code: עבור Thread שהמזהה שלו הוא i

$$\begin{aligned}&\text{flag}[\text{i}] = \text{True};\\&\text{Turn} = 1 - \text{i};\\&\text{while}\left(\text{flag}[1 - \text{i}] \quad \text{and} \quad \text{turn} == 1 - \text{i}\right);\end{aligned}$$

- Exit Code: $\text{flag}[\text{i}] = \text{False}$.

כלומר, כאשר חוט רוצה לגשת לקטע קוד קריטי, הוא יודיע שהוא רוצה על ידי הדלקת הדגל שלו, ואז ישאל את החוט האחר, האם הוא גם רוצה. במידה שכן, הוא יחכה שיסיים. במידה שלא, הוא יכנס לקטע הקוד, ובסופו יכריז שהוא לא מעוניין לגשת אליו. ניתן להוכיח שאכן התנאים מתקיימים.

בעיה. הקוד מתאים רק לשני חוטים ולא ניתן להרחבה בקלות לכמה חוטים. מלבד זאת, כאשר חוט מחכה, הוא כל הזמן בודק את התנאי של הלולאה, ולכן ה-Scheduler שמאפשר לו לבצע זאת על ה-CPU מבזבז משאבים, שכן נעדיף שמישהו יודיע לחוט שהתנאי לא מתקיים.

נגדיר מושגים קריטיים הנוגעים לאלגוריתמים הנ"ל:

הגדרה. מצב תחרות (Race Condition): מצב בו התזמון של תהליכים או חוטים (הסדר בו כל אחד מקבל את המעבד) משנה את התוצאה הסופית. כלומר תזמון שונה מביא לתוצאות שונות.

הערה. רוב המקרים, נקבל שהכל בסדר, עד שלפתע יש באג. מאוד קשה לדבג את זה.

הגדרה. פקודה אטומית (Atomic Instruction): פקודה שהמעבד יכול לבצע בלי שיתבצע באמצע Context Switch.

דוגמה. lw, sw פקודות אטומית (ברוב המקרים). $x = x + 1$ איננה פקודה אטומית.

נעיר כי לעיתים פקודת lw מכילה גם כמה פקודות, כיוון שהמילה ארוכה מדי ומתפרשת על כמה כתובות זכרון, במקרה זה היא איננה אטומית.

הגדרה. Busy Waiting (spinning, busy looping): טכניקה בה תהליך בודק באופן חזרתי ואקטיבי אם תנאי מתקיים.

4.2.2 אלגוריתם המאפיה (Bakery)

נציע דרך קלאסית לסנכרון. נחזיק כרטיסיה של מספרים, וכל פעם שחוט רוצה לגשת לקטע קוד קריטי, הוא יקח מספר, ויחכה שכל החוטים עם מספר קטן ממנו, יבצעו את קטע הקוד.

מה הבעיה? לקיחת מספר איננה פעולה אטומית ולכן אין כאן Mutual Exclusion.

למשל, חוט אחד לקח את הספרה 2, אך תוך כדי הלקיחה, גם חוט אחד לקח את הספרה 2, ולכן שניהם יגשו במקביל לקטע הקוד הקריטי.

נבהיר שכאשר אנו לוקחים מספר, אנו בוחרים את המקסימום של כל מי שלידנו, ומוסיפים אחד, אם כך,

נציע דרך למימוש:

```
number[i] = 1+max{number[j]} for j…=1,,n;
for j=1 to n {
      while(i!=j and number[j]>0 and
             number[j]<number[i]);
}
```

כלומר נעבור על כל הסביבה, כל עוד החוט שאנחנו בודקים הוא לא אנחנו, והוא רוצה לגשת לקטע הקוד הקריטי, ויש לו עדיפות על פנינו, נחכה.

נביט בפעולה הראשונה שלא לקיחת המספר. לפני שהוכנס המספר אל תוך number [i], יתכן שיתבצע context switch וחוט אחר יקבל את המספר שהוא בא לקחת. בחזרה אל החוט המקורי, הוא יקבל אותו מספר, ולכן שניהם יגשו אל קטע הקוד.

כדי לתקן את זה, נחליף את התנאי $\text{number[j]} < \text{number[i]}$ ב-$\text{number[j]} \leq \text{number[i]}$. הבעיה כאן, היא שאמנם אין גישה לקטע הקוד הקריטי ביחד, אך אין גישה בכלל, יש כאן Dead Lock!

נציע אם כך השוואה לקסיקוגרפית, כלומר

```
number[i] = 1+max{number[j]} for j…=1,,n;
for j=1 to n {
      while(i!=j and number[j]>0 and
             (number[j],j)<(number[i],i) );
}
```

כלומר, אם המספרים שווים, אז נתעדף חוט עם מזהה קטן יותר. האם זה יספיק? הבעיה טמונה בשורה הראשונה. נניח שחוט מספר 4 רוצה לקחת מספר. הוא מחשב את המספר שלו, עובר על המספר של חוט 5, רואה שהוא אפס וממשיך לעבור על כל החוטים הבאים. לפתע, מתבצע Context Switch וחוט 5 מחשב את המספר שלו, הוא עובר על המספרים, רואה שהמספר של 4 הוא עדיין אפס, ומחשב את המקסימום. הוא נכנס אל הלולאה, רואה שהוא יכול להיכנס לקטע הקוד הקריטי, שכן חוט 4 עדיין עם מספר 0, ונכנס.

עתה מתבצע שוב Context Switch וחוט 4 מסיים לחשב את המקסימום, מבחינתו, המקסימום זה המספר של 5, ולכן הוא יכנס ללולאה, יראה שיש לו את אותו מספר של 5, אבל המזהה שלו, 4 קטן מהמזהה של 5, 5, ולכן הוא יכנס לקטע הקוד הקריטי גם, ונקבל שאין Mutual Exclusion.

כלומר, עלינו להגן על חישוב המקסימום. המימוש הבא, הוא מימוש נכון:

```
Choosing[i]=true;
number[i] = 1+max{number[j]} for j…=1,,n;
Choosing[i]=false;
for j=1 to n {
          while(Choosing[j]);
      while(i!=j and number[j]>0 and
             (number[j],j)<(number[i],i) );
}
```

כלומר, לא ניתן לבצע את ההשוואה בלולאה, מבלי שחוט סיים לקבל את המספר שלו. לכן חוט 5 במקרה הקודם, לא יוכל להיכנס לפני שחוט 4 יכנס.

4.3 פקודות Read – Modify – Write

כיוון שפעולת קריאה וכתיבה מהזכרון הן פעולות חשובות, שצריך להגן על כל אחת מהן, כלומר, לא ניתן לבצע את שתיהן בלי הגנה, נכתבו פקודות ייעודיות לכך בחומרה, שכן פקודת חומרה לא ניתן לעצור על ידי Context Switch. אשר מסוגלות לכתוב ולקרוא בו זמנית. במחשבים מודרניים יש שלוש פקודיות בסיסיות:

- Test&Set ($\&lock$) - מבצעת $i = *lock$ משימה $*lock = 1$ ומחזירה את i.
- Fetch&Add ($\&p, inc$) - דומה לפקודה הראשונה, רק שאיננה בינארית: שומרת $val = *p$ מוסיפה $*p = val + inc$ ומחזירה את val.
- Compare&Swap ($\&p, old, new$) אם $*p \neq old$ היא מחזירה $false$ ואחרת, היא משימה $*p = new$ מחזירה $true$.

אנחנו נתמקד ב-Test&Set.

באמצעות פקודה זו, נציע את האלגוריתם הבא:

```
While(Test&Set(lock));
```

ולאחר ביצוע קטע הקוד הקריטי נגדיר $lock = 0$.

במקרה זה אכן נקבל Mutual Exclusion, אבל תהליכים חסרי מזל עלולים לחכות לנצח, ולכן יש כאן הרעבה.

כדי להבטיח Starvation Freedom, האלגוריתם צריך להיות הרבה יותר מסובך.

4.4 Synchronization Primitives

עד כה עינינו על תנאי המודל. אך ראינו שיש בעיות נוספות, כמו Busy Wait. הדרך להתמודד עם בעיה זו, הוא באמצעות עזרה ממערכת ההפעלה.

נגדיר עבור משימה זו מבנה נתונים אבסטרקטי, שהגישה אליו היא באמצעות הממשק הספציפי של הספריה, שמגדירה אותו במערכת ההפעלה.

צורת המימוש שלו, איננה מעניינת את המשתמש, אלא רק הממשק. למעשה, יתכן שבתוך המימוש עצמו יהיה Busy Wait.

מבנה הנתונים הראשון שנדבר עליו הוא ה-Semaphore שהומצא על ידי אדם הולנדי שבכלל שימש לתזמון רכבות.

הוא מכיל שני שדות:

- ערך (value).
- List (L).

על מבנה זה, נגדיר פעולות, שיעזרו לנו לפתור את הבעיות. היתרון במבנה אבסטרקטי, הוא שמערכת ההפעלה עוזרת לנו מתחת לפני השטח, ולכן באמצעות המבנה, נוכל להחזיר תהליכים למצב Ready/Blocked, הכל כמובן, באישור מערכת ההפעלה מתחת לפני השטח.

4.4.1 Semaphore

הסמפור הוא מבנה נתונים של מערכת ההפעלה לסנכרון threads, הוא פועל כמו מנהל משמרת במסעדה:

נניח כי יש תור של כמה אנשים, המחכים לשבת במסעדה. בהתחלה, יש לנו v מקומות פנויים, ולכן אנו נותנים מקום ל-v קבוצות ישיבה, אבל לאחר מכן, אנו נותנים לכל התור לחכות, עד שקבוצה תסיים לאכול. כלומר, אנו מכניסים את כל הקבוצות הממתינות למצב "שינה" Busy Wait. את זאת ניתן לממש עם שני שדות:

- ערך (value).
- List (L).

ושלוש פעולות:

- הכנסת thread למצב שינה:

```
Down(S)
        S.value = S.value - 1
        if S.value <  0 then
                { add this thread to S.L;
                  sleep();
        }
```

- שחרור thread, והעברתו למצב Ready, כדי שה-Scheduler יוכל להעביר אותו למצב running:

```
UP(S)
        S.value = S.value + 1
        if S.value <=  0 then
                { remove a thread T from S.L;
                        Wakeup(T);
        }
```

- ואתחול:

```
Init(List: S, integer: v)
        S.value = v
```

שתי הפעולות Down, Up הן פעולות אטומיות, ולכן איננו צריכים לדאוג מהגנה עליהן בפני כמה threads שונים. המימוש, הוא בעיה של מערכת ההפעלה, שבמחינתה, יכולה לחסום את כל הסיגנלים בעת ביצוע קטעי הקוד, כך שבפועל לא באמת מדובר בקטעי קוד אטומיים, אלא רק פרקטית, כלומר, לא ניתן לבצע Context Switch בעת ביצוע הפעולות.

הערה. Semaphore נקרא הוגן אם התור שהוא מחזיק הוא First in first out, ולא הוגן, אם הוא לא מחוייב לכך.

Mutual Exclusion עתה, בהנתן Semaphore, נוכל להגן על קטע קוד קריטי באופן הבא:

Remainder
down (lock)
Critical
Up (lock)

אנו מניחים שה-Semaphore, lock, הוא משותף לכל החוטים שרצים. כאשר הערך המקסימלי של האיברים ברשימה הוא 1 והערך המינימלי הוא 0, מדובר ב-Semaphore בינארי, והוא שקול לחלוטין ל-Mutex.

דוגמה. Semaphore מסוגלים להכריח תכנית לבצע פעולות בסדר מסוים, דבר ש-Mutex לא בהכרח יכול, שכן הוא רק מגן על קטע קוד קריטי. כדי לעשות זאת, נניח שיש לנו קטעי קוד A, B וסמפור flag. על כן, בהנתן שני תהליכים, נבצע

...	Down (flag)
A	B
UP (flag)	...

מה עשינו כאן? תחילה הסמפור מאותחל עם $\text{value} = 1$, ולכן לאחר ביצוע down הוא יכיל את הערך 0, ויכניס את התהליך השני למצב שינה. בינתיים תהליך 1 יריץ את קטע הקוד A, יעשה UP (flag), הערך $\text{value} = 1$, ואז תהליך 2, יריץ את קטע הקוד B. ככה למעשה הכרחנו את התהליכים לרוץ בסדר כרצוננו.

שאלה האם Semaphores פותרים את כל בעיות הסנכרון?

דוגמה. נניח שחוט A רוצה להעביר כסף לחוט B וחוט B רוצה להעביר כסף לחוט A.

ראינו שכאשר חוטים מבצעים $++, --$ במקביל, דברים מוזרים קורים. לכן כדי להבטיח שהפעולות יקרו כמו שצריך נבצע

down (lock)	down (lock)
Move A $\to$ B	Move B $\to$ A
up (lock)	up (lock)

אבל, מה יקרה כאשר יהיו לנו שלושה תהליכים כאשר כל אחד יעביר כסף לשני, כלומר $A \to B \to C \to D$? הפתרון של חסימת כל גישה מקבילה, לא יעיל, כלומר

down (lock)	down (lock)	down (lock)
Move A $\to$ B	Move B $\to$ C	Move C $\to$ D
up (lock)	up (lock)	up (lock)

שכן כאשר $A \to B$ מדוע שלא נעביר גם $C \to D$? על כן, נוכל להשתמש בשני Semaphores:

down (SA)	down (SB)	down (SC)
down (SB)	down (SC)	down (SD)
Move A $\to$ B	Move B $\to$ C	Move C $\to$ D
up (SA)	up (SB)	up (SC)
up (SB)	up (SC)	up (SD)

כלומר, אנו מוודאים שיש גישה בטוחה לשני חשבונות הבנק הרלוונטים בלבד, ורק לאחר מכן ניגשים אליהם. במקרה זה $C \to D$ יכול להתבצע במקביל ל-$A \to B$. האם יש כאן DeadLock? דווקא לא, אבל במקרה הבא:

down (SA)	down (SB)	down (SC)	down (SD)
down (SB)	down (SC)	down (SD)	down (SA)
Move A $\to$ B	Move B $\to$ C	Move C $\to$ D	Move D $\to$ A
up (SA)	up (SB)	up (SC)	up (SD)
up (SB)	up (SC)	up (SD)	up (SA)

מה קורה כאן? 1 מחכה ל-SB, 2 מחכה ל-SC, 3 מחכה ל-SD אבל 4 מחכה ל-SA וקיבלנו DeadLock. האם יש דרך לפתור זאת?
נוכל לנעול את המשאבים לפי סדרם, ולא לפי המקור והיעד, כלומר

down (SA)	down (SB)	down (SC)	down (SA)
down (SB)	down (SC)	down (SD)	down (SD)
Move A $\to$ B	Move B $\to$ C	Move C $\to$ D	Move D $\to$ A
up (SA)	up (SB)	up (SC)	up (SD)
up (SB)	up (SC)	up (SD)	up (SA)

הפעם, אם תהליך 4 לא מחכה ל-SA, אזי או שהוא מריץ, או שתהליך 3 לקח את SD ולכן בכל מקרה יש חוט שרץ. אחרת, אם תהליך 4 מחכה ל-SA זה אומר שחוט אחד לקח אותו, ולכן אם הוא מחכה ל-SB, אזי אם חוט 2 מחכה ל-SC זה אומר שחוט 3 רץ, אחרת חוט 2 רץ ואחרת הוא לא מחכה ל-SB כלומר חוט 1 רץ. לכן בכל מקרה יש חוט שרץ.

בעיית סנכרון האכילה במקלות האכילה נהוג למדל את מה שתיארנו בדוגמא הקודמת, באופן הבא. נניח שיש לנו 6 פילוסופים שיושבים במעגל, כאשר ביניהם יש מקלות אכילה, כך שמימין ומשמאל לכל אחד, יש מקל אכילה בלבד. לקיחת המקל היא לפי מה שבחר בו ראשון.
על מנת למדל זאת בשפת מערכת ההפעלה, כל פילוסוף הוא תהליך, כל מקל אכילה הוא Semaphore, המגן על קטע הקוד הקריטי שהוא למעשה "אכילה". כיצד נסנכרן אותם? נמספר אותם ב-$0, \ldots, 5$ ונגדיר

```
down (chopstick [(i + 1)  mod 6])
down (chopstick [i])
Eat
Up (chopstick [(i + 1)  mod 6])
Up (chopstick [i])
Think
```

האם זה יעבוד? יש כאן DeadLock בגלל הסימטריה. לכן, כדי לטפל בזה, נשבור את הסימטריה! נחליף את קטע הקוד

```
down (chopstick [5])
down (chopstick [0])
Eat
Up (chopstick [0])
Up (chopstick [5])
Think
```

טענה. עם השינוי שהוצע, אין בסכימה DeadLock.

הוכחה. נניח בשלילה שיש DeadLock, כלומר שפילוסוף i מחכה למקל, וכל השאר לא אוכלים. בה"כ i הוא $0, 1, 2$ או 3. אם הם מנסים לקחת את המקל השני, זה אומר שמי שלידם, ניסה לקחת את הימני והצליח, אבל כיוון שקודם לוקחים את המקל השמאלי, הוא לקח את שניהם ולכן הוא אכל, סתירה!

אם $i = 4$, אז אם הוא לקח קודם את מקל 5, ולא מצליח, זה אומר ש-5 לקח אותו, ומחכה למקל 0, אותו לקח פילוסוף 0, אבל מהשינוי שהוצע, קודם לוקחים את 5, ולכן מחכים לאפס, מה שאומר ש-0 לקח את המקל השני שלו, ולכן הוא רץ, סתירה.

אם $i = 5$, אז אם הוא לקח את מקל 0 ולא מצליח זה אומר ש-0 לקח את 0 ולכן הוא נכנס. סתירה. □

האם יש כאן Starvation freeedom? זה תלוי בהאם ה-semaphore הוגן או לא.

תנאים הכרחיים לקיום Dead Lock באופן כללי, יתכן כי יש dead lock אם ורק אם מתקיימים התנאים הבאים:
Mutual Exclusion: לפחות משאב אחד ננעל, כלומר יש קבוצה שמחכה לו.
Hold and Wait: חוט אחד נעל משאב אחד, ומחכה למשאב אחר, שננעל על ידי חוט אחר.
$\text{Non} - \text{Preemptive Allocation}$: משאב יכול להשתחרר אך ורק על ידי הגורם שנעל אותו.
Circular Wait: חוט אחד מחזיק במשאב, חוט אחר מחכה לו ומחזיק במשאב אחר, חוט נוסף מחזיק את המשאב הנ"ל ומחכה למשאב נוסף שמוחזק על ידי חוט נוסף וכן הלאה...

נהוג להסתכל על תנאים אלה כגרף - הקודקודים הם המשאבים והתהליכים. חץ שיוצא ממשאב לתהליך מסמן שהמשאב ננעל על ידי התהליך, וחץ שיוצא מתהליך למשאב מסמן שהתהליך מחכה למשאב. במידה שיש מעגל בגרף, תנאי ההמתנה המעגלית מתקיים.

הערה. בשבירת הסימטריה שביצענו קודם בבעיית הפילוסופים, הייתה לנו מקביליות מאוד חלשה, כלומר במקרה הגרוע רק אחד אכל. בפועל, אפשר לסנכרן את הפילוסופים האי זוגיים באופן דומה, ואת הזוגיים באופן הפוך, ואז לפחות קבוצה אחת תוכל לאכול במקביל. זו מקביליות טובה בהרבה.

מסקנה. אם ננעל משאבים לפי סדר המשאבים, ננעל אותם תמיד באותו הסדר, ונשחרר אותם תמיד בסדר הפוך, לעולם לא נקבל מעגל בגרף, ולכן לא יהיה dead lock.

Cyclic Spooler ראינו שהספולר ממומש על ידי Buffer, ושכאשר הפעולות עליו לא אטומיות, או לא מוגנות, יתכנו כל מיני באגים משונים כמו הדפסה של אותה הודעה פעמיים, אי הדפסה של הודעה ועוד. הגודל של ה-Buffer מוגבל, ולכן כדי להרחיב את קיבלת האכסון, הרבה פעמים משתמשים ב-Buffer ציקלי, בו כאשר מגיעים לסוף התור, קופצים להתחלה. דבר זה עלול להביא הרבה בעיות, אם לא ממשים זאת נכון. נביט למשל בשני המצבים הבאים של ה-Buffer:

$$\begin{array}{ccc} & H & H \\ out \rightarrow & B & B \leftarrow out \\ in \nearrow & C & C \nwarrow in \\ & D & D \\ & E & E \\ & F & F \\ & G & G \end{array}$$

על פניו, נראה שמדובר באותו מצב, אך איך נדע אם הוא מלא, או ריק? באותה מידה יתכן שהוא מלא או ריק, ולכן צריך לשמור אינדיקטור שיענה על השאלה הזו. בלעדיו, לא נדע אם להדפיס או לחכות. דרך לקבל אינדיקטור כזה היא שמירת מספר העבודות שנותרו להדפסה. על כן, המכניס לתור והמוציא, צריכים לעדכן אותו בהתאם. כמובן, העדכון צריך להיות אטומי, או מוגן. יחד עם זאת, כדי לאפשר גישה לספולר עבור כמה תהליכים במקביל צריך לעשות שימוש ב-Semaphores:

Producer	Consumer
down (empty)	down (full)
down (mutex)	down (mutex)
buffer [IN] = job;	job = buffer [OUT] ;
IN = IN + 1 mod n;	OUT = OUT + 1 mod n;
UP (mutex)	UP (mutex)
UP (full)	UP (empty)

כאשר מאותחלים: $\text{mutex.val} = 1, \text{empty.val} = \text{n}, \text{full.val} = 0$. ניתן לראות שכאשר המדפיס רוצה להדפיס הודעה כשהתור ריק, הוא נעצר, בגלל ש-$\text{full.val} = 0$. כאשר מגיעה עבודה, מורידים את מספר התאים הריקים, ומגנים על קטע הקוד באמצעות ה-mutex.

4.4.2 Monitoring

מבנה נתונים נוסף לסנכרון הוא Monitor. הוא מבנה שנחשב ממש כאובייקט, ואחראי לחלוטין על הסנכרון. למשל, כל הוספה ל-Buffer של הספולר תתבצע על ידי פעולות של המוניטור. לכן, הוא מקל מאוד על המשתמש, ולרוב משתמש ב-Semaphores מאחורי הקלעים.

המוניטור מבטיח Mutual Exclusion בצורה כזו, שמהלך חייו, רק חוט אחד יכול לגשת אליו במקביל. זה תנאי מאוד חזק, כיוון שהוא חוסם את כל החוטים שמנסים לגשת אליו פרט לאחד ואת כל קטעי הקוד שהוא מכיל. לכן אנו מקבלים את התנאי בחינם.

הבעיה העיקרית שעולה מכך היא חוסר מקביליות. למשל בהנתן שני Producers (p) , consumers (c), כאשר c_1 יגש ל-spooler הוא יעצור את c_2, p_1, p_2 וזה בעייתי, כי אין סיבה ש-p_1 לא יגש לקטע הקוד של ה-producer. לכן, הוסיפו ל-monitor פעולות חדשות.

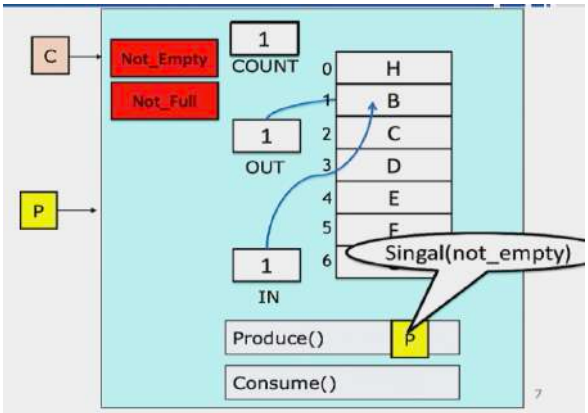
Condition Variables במקום שכולם יחכו בתור, כולם יחכו שתנאי מסויים יתקיים, וכשהוא יתקיים יודיעו להם, והם ינסו לגשת לקטע הקוד. על כן, הוגדרו שלוש פעולות עיקריות:

(wait) wait : משחרר נעילה של מוניטור כך שחוט נכנס, ומחכה לתנאי שיתקיים. כלומר, יש תור של חוטים שמחכים להיכנס ולקבל סיגנל.

(notify) signal : שליחת סיגנל לחוט שהוא יכול לגשת לקטע הקוד, כלומר מעירים חוט ישן. אם אין חוט כזה, הסיגנל נאבד. (אין היסטוריה של סיגנלים שנשלחו).

(notifyall) broadcast : מעיר את כל החוטים הישנים.

במקרה שתיארנו קודם, יחד עם התנאים הנ"ל נקבל את הסכימה הבאה:



איור 23: במקום ש-P_1 יחכה ל-C_1, C_1 נשלח לתור של המשתנה NotEmpty. P_1 מגיע לקטע הקוד שלו, ושולח סיגנל של notifyAll על המשתנה NotEmpty, ואז C_1 מבצע את קטע הקוד שלו.

הערה. תקשורת בין חוטים מתבצעת על ידי זכרון משותף, או על ידי שליחת הודעות בערוצי תקשורת. אלגוריתמים שמקשרים חוטים באמצעות ערוצים אלה, נקראים אלגוריתמים "מבוזרים", יש להם בעיות דומות, אך מעט שונות.

5 תזמון Scheduling

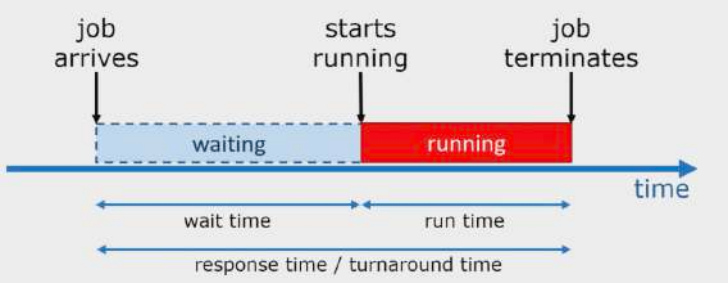
5.1 מבוא

פעולות תזמון נמצאות בכל מקום, למשל תזמון קריאה מדיסק מסתובב, כך שהראש יסתובב כמה שפחות מצד לצד, פקטות ברשתות תקשורת, בקשות בשרתי Web.

אם ניזכר בסכימה שראינו לתהליכים, אנו עוסקים בפעולות ה-Schedule, Preempt: מי שיעניינו אותנו הם:

- CPU Scheduler - מי שמחליט איזה תהליך רץ.
- Dispatcher - מי שמבצע את פעולות ה-CPU.

להל"ן חלוקה לקטעי זמן חיים של התהליך:



איור 24: זמני החיים של התהליך.

5.2 Scheduler

המתזמן הוא אלגוריתם:

קלט העבודות שצריכים לתזמן.

פלט החלטה את מי להריץ עכשיו.

מטרה אלגוריתם יעיל לתזמון. (לא נרצה שהוא יפתור בעיות $\text{Hard} - \text{NP}$, ונרצה שהוא יהיה בעל ביצועים טובים ביחס למערכת.

מודל חישובי מה הפעולות שאפשר לעשות? האם מותר לו לבצע Preempt?

שאלה מהי פונקציית המטרה?

נרצה פונקציה שתגיב מהר לתהליכים חדשים שצריך לתזמן, כלומר בעלת Response Time מינימלי. כיוון שיש הרבה תזמונים, נרצה שהממוצע "התוחלת" תהיה מינימלית.

נציג כמה מדדים לביצועי אלגוריתם Scheduling.

הגדרה. $\text{Low} - \text{Response Time}$ סך כל הזמן לביצוע פעולה $(\text{Wait Time} + \text{Run Time})$. (מה שאכפת למשתמשים).

הגדרה. $\text{Low} - \text{Wait Time}$ מה שנשלט על ידי המתזמן.

הגדרה. $\text{Low} - \text{SlowDown}$ מוגדר על ידי $\frac{\text{Response Time}}{\text{Run Time}}$.

שאלה מה אפשר לומר על הגודל הנ"ל?

תשובה הוא חסום על ידי 1, וככל שיש יותר תגובה, כך הוא גדל. לכן נרצה שהוא יהיה כמה שיותר קרוב ל-1.

מלבד מהירות, נרצה שהתזמון יהיה הוגן. כלומר, כל אחד יקבל את מה שהוא ראוי לקבל, זה לא בהכרח שווה. למשל, נחליט שפרופסורים מקבלים 70% מכוח החישוב באוניברסיטה.

5.3 Off/On Line Algorithms

הגדרה. אלגוריתם $\text{Off} - \text{line}$ הוא אלגוריתם שמקבל את כל הקלט בתחילת התכנית.

הגדרה. אלגוריתם $\text{On} - \text{line}$ הוא אלגוריתם שמקבל חלקים מהקלט במהלך ריצת התכנית, ומבצע פעולות בהסתמך על חלקים אלו.

5.3.1 מודל $\text{Off}-\text{line}$

נניח כי יש לנו רשימה סופית של עבודות עם זמן ריצה לכל עבודה.

בהנתן אלגוריתם זה, המודל הפשוט ביותר הוא שימוש ב-$\text{FCFS}-\text{First Come First Serve}$. למודל זה נוכל לחשב זמן המתנה ממוצע, זמן תגובה ממוצע וכדומה.

מה הבעיה במודל זה? עבודות מהירות עלולות "להתקע" מאחורי עבודות גדולות. כדי לטפל בזה נוכל לתת לעבודה מהירה להתבצע לפני עבודה ארוכה יותר.

למעשה, ההחלפות הנ"ל מבצעות Bubble Sort על תור העבודות לפי זמני העבודה, ולכן אפשר למדל זאת בצורה מעט יותר מוצלחת:

ה-Shortest Job First (SJF), הוא מודל בו ממיינים את רשימת העבודות, וניתן לעשות זאת באמצעות אלגוריתמים יעילים יותר מ-Bubble Sort. כמו כן, הוא משפר מאוד את המדדים.

מטרה ה-SJF ממזער את ממוצע זמן ההמתנה.

שאלה האם יש אלגוריתם טוב יותר מה-SJF?

תשובה נטען שלא.

טענה. לא קיים אלגוריתם שנותן זמן המתנה ממוצע קטן יותר מפלט ה-SJF.

הוכחה. נניח שמתזמן S הוא בעל זמן המתנה ממוצע מינימלי. אם S הוא לא ה-SJF אזי קיים זוג עבודות כך ש-$p_i.\text{runtime} > p_{i+1}.\text{runtime}$, על כן, החלפת העבודות מקטין את התרומה לממוצע זמן ההמתנה, מבלי לשנות שום דבר אחר. מכאן אנו מקבלים סתירה לכך ש-S אופטימלי. □

שאלה האם שימוש ב-Preemption יעזור?

תשובה כיוון שהכל ידוע מראש, ואנו מחליטים לתזמן הכל כרצוננו, שינוי Online לא ישנה, שכן מראש היינו יכולים לתזמן זאת בהתאם.

שאלה מה לגבי זמן תגובה? האם הוא גם אופטימלי?

תשובה הוא גם אופטימלי שכן $\text{Response Time} = \text{Wait Time} + \text{Run Time}$ ולכן הממוצע הוא סכום הממוצעים. כיוון שממוצע ה-$\text{run}-\text{time}$ הוא קבוע, מזעור ה-$\text{Wait}-\text{Time}$ ממזער את ממוצע ה-Response Time.

5.3.2 מודל $\text{On}-\text{line}$

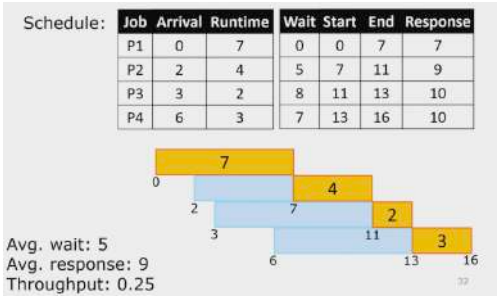
בשונה ממודל ה-$\text{Off}-\text{line}$ כאן העבודות לא ידועות מראש.

המודל הראשון שנשתמש בו דומה למודל הקודם, אך ההבדל הוא שעבודות עם זמנים לא ידועים מראש, מגיעות מבחוץ. מה האלגוריתם הכי פשוט שאפשר לחשוב עליו?

תשובה שוב FCFS.

כיוון שעבודות חדשות מגיעות לאורך הריצה, כל פעם שעבודה מגיעה, צריך לשאול מה אמורים לעשות? האם צריך לשנות את סדר הביצוע? או שעלינו לבצע לפי סדר ההגעה.

כאשר המעבד עובד, אין סיבה לשאול מה עושים, כיוון שאין עבודה שצריך לבצע. אך ברגע שהוא מתפנה צריך להחליט!

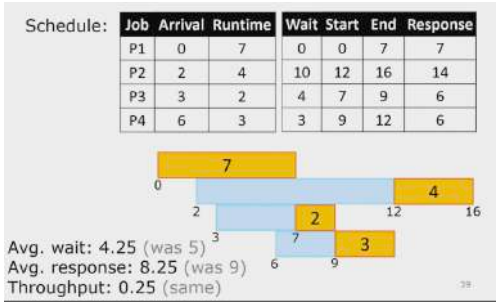


Job	Arrival	Runtime	Wait	Start	End	Response
P1	0	7	0	0	7	7
P2	2	4	5	7	11	9
P3	3	2	8	11	13	10
P4	6	3	7	13	16	10

איור 25: דוגמא לריצת אלגוריתם ה-Online

האם ניתן לשפר זאת?

באלגוריתם ה-OFFline הצענו את ה-SJF, שלו מתאים אלגוריתם $\text{On}-\text{line}$. עבור הדוגמא הקודמת נקבל:



איור 26 : דוגמא לריצת אלגוריתם ה-$\text{SJF} - \text{Online}$

הגדרה. $\text{PriorityScheduling}$ - כל אלגוריתם ניתן לייצג לפי סדר עדיפויות.

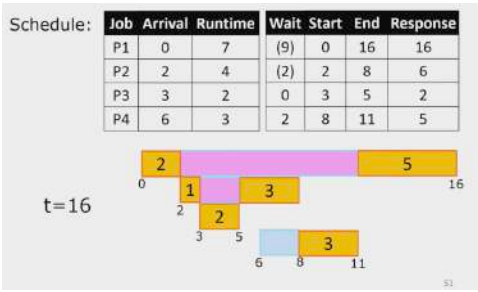
דוגמה. ב-FCFS, העדיפות היא הזמן שעבר מזמן ההגעה. ב-SJF העדיפות היא זמן הריצה.

בעיה. ל-SJF יש בעיה. כאשר P_1 מגיעה, אנו תקועים איתה עד שמשימה חדשה תגיע. אנו לא יודעים את העתיד, ולכן אנו פשוט מבצעים אותה.

שאלה כיצד נפתור את הבעיה?

תשובה נשתמש ב-Preemption! זה יפצה על העובדה שאנו לא יודעים את העתיד, אך המחיר יהיה של-Context Switching יש Overhead.

מכאן נקבל את המודל הבא. בכל פעם נבחר את התהליך שזמן הריצה שנותר לו הוא הקצר ביותר. אנו בוחרים ב"זמן הריצה שנותר" הקצר ביותר, כיוון שיתכן, שכפי שתואר, תהליך נבחר, אך תהליך חדש הגיע עם זמן ריצה קטן יותר, ולכן עדיף להריץ אותו.
עם מודל זה על הדוגמא שראינו אנו מקבלים :



איור 27 : דוגמא לריצת אלגוריתם ה-$\text{Preemption} - \text{SJF} - \text{Online}$

בעיה. אנו לא יודעים כמה זמן לוקח לתהליך לרוץ! אנו לא יודעים שום דבר על העתיד.

שאלה כיצד נפתור את הבעיה?

תשובה נשערך את זמן הריצה. כלומר נבנה מודל סטטיסטי.

דוגמה. נסמן זמני ריצה באופן הבא :

1. t_n זמן הריצה של העבודה ה-n.
2. τ_{n+1} זמן הריצה שנחזה לעבודה הבאה על ה-CPU.
3. $0 \leq \alpha \leq 1$ פרמטר ממשקל.
4. $\tau_{n+1} = \alpha t_n + (1-\alpha)\,\tau_n$. אנו נותנים משקל לעבודה הקודמת, ולתחזית הקודמת. זו תחזית אקספוננציאלית דועכת, שכן יש כאן תלות בחזקות של בסיס קטן מ-1. (תנאי ההתחלה נתון לבחירתנו)

יש כאן הרבה overhead ויתכן שלא נקבל תוצאות טובות, אך זו דרך לתחזית, שכן יוצרת עדיפת לפרמטרים שונים.

היות שאנו לא מסוגלים לדעת כמה זמן לוקח לכל תהליך לרוץ, בוא נניח שאנו יכולים! (זו הנחה מתמטית קלאסית). למשל, נניח שכל העבודות יכולות לרוץ ביחד על המעבד באופן הנקרא "Process Sharing", למשל, כאשר k עבודות מגעיות, כולן מתקדמות בקצב ריצה של $\frac{1}{k}$.
כיצד זה משפיע על התזמון?
כלומר בכל קיום התכנית, כל העבודות רצות, וכל פעם שעבודה מסתיימת, קצב הריצה גדל.

יתרונות עבודות קצרות לא נתקעות.

חסרונות כולן רצות לאט יותר.

מסקנה. המדדים תלויים בזמני הריצה של העבודות.

המקרה הטוב הוא כאשר יש הרבה עבודות קצרות, ואז עבודה ארוכה שתוקעת עבודות קצרות, לא תוקעת אותן אלא רק מעכבת, ולכן זה מעין קירוב "צולע" ל-SRPT.

המקרה הגרוע, הוא כאשר יש הרבה עבודות באותו האורך, שכן כולן רצות ביחד, ולאט. במילים אחרות, ה-Entropy מאוד נמוכה.

מסתבר ש-Process Sharing טוב כאשר התפלגות זמני הריצה מוטה, כלומר בעלת Entropy גבוהה. בפרט, המדד

$$\text{CoefficientVariation} = \text{CV} = \frac{\text{std\_devitation}}{\text{mean}} > 1$$

כלומר, כאשר ה-$\text{std\_devitation} > \text{mean}$ אנו אומרים שההתפלגות "מוטה". במקרה זה כאמור, התזמון טוב.

מלבד זאת, ניתן להסתכל על זה לפי הנתונים, ולהסתכל על ההתפלגות באמת. מסתבר שהתפלגות הזמן של התהליכים מקיימת את הקשר

$$\Pr[r > t] = \frac{1}{t}$$

כלומר הזנב דועך פולינומיאלית, ולכן יש הסתברות לא זניחה לראות ערכים מאוד גבוהים, ולכן ההתפלגות מוטה. נעיר כי כאשר זנב ההתפלגות דועך אקספוננציאלית, ההתפלגות כנראה לא מוטה, שכן ערכים גדולים מתקבלים בהתפלגות זניחה.

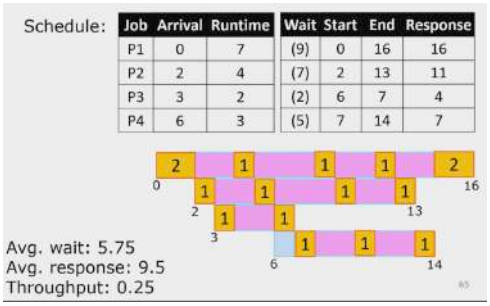
על כן, אנו מקבלים כי Process Sharing אמור לעבוד טוב.

בעיה. לא ניתן לבצע באמת Process Sharing, שכן מעבד יכול להריץ פעולה אחת בלבד ולא כמה פעולות במקביל!

5.3.3 אלגוריתם Round Robin

כדי לממש Process Sharing, הוצע האלגוריתם המקרב של Round Robin.

אנו נריץ כל עבודה לזמן קצר שיוגדר מראש (qunatum), ואז נבצע Context Switch לעבודה אחרת. יש לנו כאן טרייד-אוף בין ה-Qunatum לבין ה-Context Switch.



איור 28 : דוגמא לריצת קירוב ל-$\text{Process} - \text{Sharing}$, עם האלגוריתם $\text{Round} - \text{Robin}$

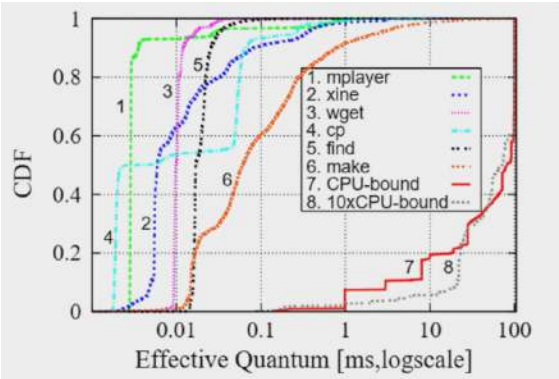
הפרמטר החשוב ביותר לאלגוריתם הוא q.

נבחין כי כאשר יש n עבודות, כל עבודה מחכה לכל היותר $q\,(n-1)$. מכאן q קטן גורר זמן המתנה קטן יותר.

עתה, אם Context Switch לוקח זמן c, יש לנו Overhead שהוא $\frac{c}{q+c}$. לכן q קטן גורר Overhead גדול.

אם q גדול, עבודות יוכלו להסתיים לפני סוף ה-Quantum, מה שמוביל להתנהגות של FCFS. זה יכול להיות טוב, שכן אנו רוצים לשבור עבודות ארוכות.

הבא מתאר מדידות של פונקציית ההתפלגות המצטברת כתלות בקוואנטום אפקטיבי (כמה זמן לקח עד שתהליך הפסיק לרוץ, עקב Context Switch/terminate, לאו דווקא עקב סיום ה-Quantum) :



איור 29 : ניתן לראות שהפקודה make צריכה $q = 0.1$ כדי שתשתמש בלפחות חצי ממנו.
לולאה אינסופית (CPU Bound) משתמשת בחלק מאוד קטן מה-Quantum. מדוע? בגלל Interrupts של כרטיס הרשת, של הדיסק ועוד!

שאלה כיצד נממש Round Robin?

תשובה נשתמש ב-Timer שישלח Interrupt/Signal כל זמן מוגדר מראש, שאחריו נבצע החלפה.

הערה. $RR - Round\ Robin$ נותן עדיפות אחידה לכל העבודות.

שאלה האם ניתן לעשות משהו יותר טוב מזה?

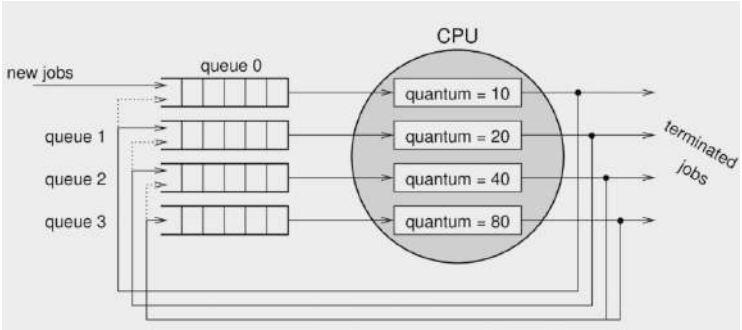
5.4 למידה מהעבר Accounting Data

נבחין כי אם הפסקנו את ריצת אלגוריתם $Round - Robin$, בגלל שהגמר הקוונטום הנוכחי, זה אומר על העבודה שהיא איננה קצרה. כלומר איננה קצרה מהקוונטום. מה נוכל לעשות עם המידע הזה?

נדע לתת לו עדיפות נמוכה.

5.4.1 $Multi - Level$ Feedback Queues

הדרך לתת עדיפות לכל תהליך רץ היא שימוש במבנה נתונים של תור. כאשר עבודה מסתיימת, לא נחזיר אותה לתור העבודות שצריכות לרוץ, אלא לתור נפרד, עם עבודות ארוכות אחרות. אנו נמשיך להריץ אותן אך ורק אם התור הראשון ריק. נוכל להכליל זאת לכמה תורים שונים לפי רמת עדיפות משתנה, ועקרונות תזמון שונים.



איור 30 : המחשה לתהליך העברת התהליכים לתור. תחילה ירוץ תהליך בעל זמן ריצה 10. אחריו יבחר תהליך עם זמן ריצה 20, הוא יעצר, ויעבור לתור הבא. ככה נמשיך עם כל התורים, עד שנגיע לתור האחרון. **תהליכים באותו התור מתוזמנים לפי אלגוריתם** Round Robin. מספר התורים במערכת הוא חזקה של 2.

בעיה. היות שתהליכים קצרים מקבלים עדיפות גבוהה, יתכן שתהליכים בעלי עדיפות נמוכה בכלל לא ירוצו.

פתרון. ניתן אחוז מענה לכל תור. למשל $60, 30, 7, 3$.

פתרון. פתרון נוסף הוא שימוש ב-Aging. כלומר ככל שתהליך מחכה יותר זמן, ככה העדיפות שלו עולה.

דוגמה. ה-Scheduler של Unix עובד באופן הבא.

ראשית הוא שומר 128 תורי עבור 128 רמות עדיפות. $0 - 49$ התורים הראשונים הם עבור הקרנל, $50 - 127$ עבור המשתמש.

המערכת תמיד מתזמנת תהליכים בעלי עדיפות גבוהה (כלומר בעל ערך נמוך). עולה השאלה, כיצד מתזמנים תהליכי משתמש?

מגדירים

$$\text{User Priority} = \text{cpu\_usage} + \text{base}\,(= 50)$$

כאשר $\text{base} = 50$ כיוון שמדובר בתהליך משתמש ולכן הוא לפחות בתור 50. התחזוק נעשה על פי הפסיקות של השעון. כל מאית שנייה, נשלחת פסיקה מהשעון, ומתווסף 1 ל-cpu_usage שנשמר ב-PCB של התהליך הנוכחי.

כאשר תהליך מפסיק לרוץ, נגיד, בגלל System Call, עוברים לתהליך הבא על פי העדיפויות שבתורים. המשקול של הזמן שתהליכים מחכים, נעשה בצורה מחזורית: בכל שנייה, עוברים על כל התהליכים שמחכים, ומחלקים ב-2 את ה-cpu_usage, ככה מעלים את העדיפות. ככה למעשה אנחנו מטפלים ב-Starvation.

גישה שונה היא שהרעבה היא לא מצב רע. הרי אם עדיפות של תהליך חדש גבוהה יותר עדיף להריץ אותו. עלינו להבחין כי מחשבים הרבה פעמים עלולים להיות במצב של overhead כלומר מצב בו שכמות העבודה שהמחשב יכול לבצע לא עולה על כמות העבודה שהוא צריך לעשות ברגע נתון. כלומר, יש תהליכים שמחכים. במצב זה, תמיד יהיו תהליכים שמחכים ולכן עדיף שתהליכים איטיים ישלמו, הרי אם תהליך אמור לקחת 13 דקות ובמקום זאת יקח לו 26 דקות, זה סביר. אבל אם תהליך אמור לקחת שנייה ולקחו לו 13 דקות, זה לא סביר בכלל. לכן מצב של הרעבה הוא לא תמיד דבר רע.

עדיפויות של תהליכים

1. תהליכים חישוביים מקבלים עדיפות נמוכה.
2. תהליכים אינטאקטיבים עם המשתמש (למשל עורך טקסט) מקבלים עדיפות גבוהה, וירוצו מיד לאחר שהם מוכנים.
3. תהליכים (מורכבים) אינטראקטיבים עם המשתמש, למשל משחקי תלת מימד, יקבלו עדיפות נמוכה, שכן הם דורשים המון כוח חישוב, ולוקחים הרבה זמן להרצה.
4. מתזמנים מודרניים שמים לב איזו אפליקציה קשורה לחלון הפעיל במסך. אפליקציה בחלון הפעיל תקבל עדיפות גבוהה, שכן אנחנו משתמשים בה.

5.5 אומדן ה-Scheduler

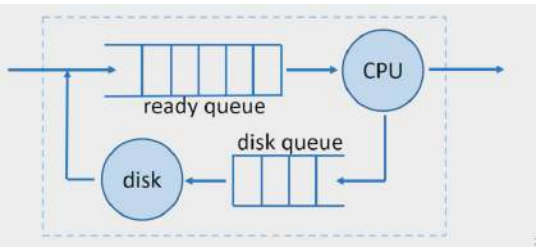
כיצד נוכל לאמוד את איכות המתזמן? אלגוריתמים קלאסיים נמדדים על פי זמן הריצה שלהם. אנחנו לעומת זאת מודדים את המתזמן על פי התוצאה הסופית. ה-Overhead של פעולת האלגוריתם פחות מעניין אותנו.

דרך נאיבית היא להריץ את האלגוריתם הרבה פעמיים ולקחת ממוצע על כל התוצאות. אבל זה יקר, כי נדרשות המון הרצות. על כן, פותחו שיטות מעט יותר טובות. ביניהן:

- Queue Models - שימוש בתורים להסקה על ממוצע התוצאות של האלגוריתם.
- Simulation - (עבור אלגוריתמים מורכבים יותר) שימשו בתכנית למימוש מודל של מערכת מחשוב, ובאמצעותה לבדוק את האלגוריתם בסימולציה.
- Implementation - הרצת האלגוריתם במערכת אמיתית, כדי לקבל את התוצאות "על אמת".

אנחנו נדבר על QueueModels.

5.5.1 הסקה באמצעות תורים - Queue Models



איור 31: יש לנו מודל, תור של תהליכים, תור של תהליכים המחכים לדיסק.

בהנתן מודל זה, אנו בוחרים תזמון למשל FCFS, כאשר הזמנים מתפלגים אקספוננציאלית כמו שראינו. דבר זה יתן לנו את ממוצע זמן התגובה.

אנחנו נדבר על הגרסא הפשוטה ביותר. נניח שיש לנו שרת אחד עם תורים. התהליכים מתפלגים פואסון λ, כלומר בממוצע מגיעים λ עבודות בשנייה. זמן מתן השירות מתפלג אקספוננציאלית μ, שהממוצע שלו הוא שאפשר לבצע μ עבודות ביחידת זמן, במילים אחרות, μ הוא קצב העיבוד של המעבד.

האם יש קשר בין λ, μ? נבחין כי אם $\lambda > \mu$ אז המערכת לא יציבה, שכן מגיעות יותר עבודות ממה שהמעבד יכול לעבד. אם $\lambda << \mu$ אז התוצאות שנקבל לא אומרות כלום על על זמן התגובה הממוצע, שכן אחוז העומס על המעבד הוא נמוך, וכל העבודות מסתיימות.

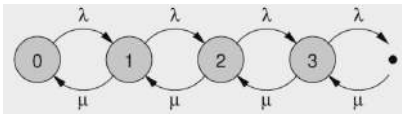
על כן, בהנתן עבודות שמגיעות בזמנים בהתפלגות פואסונית, וזמן עבודה של המעבד μ, נרצה לדעת מהו $\overline{r}$ זמן התגובה הממוצע.

כדי לנתח את זה נשתמש במשפט Little.

משפט. (Little's Law) בהנתן λ, μ (קצב הגעה וקצב עיבוד), נרצה למצוא את $\overline{r}$. מתקיים כי $\overline{n} = \lambda \cdot \overline{r}$ כאשר $\overline{n}$ פונקציה של λ, μ.

מכאן עלינו לחשב את $\overline{n}$.

כדי לעשות זאת, נחשוב על המערכת כשרשרת מרקוב. כל מצב מסמן את מספר התהליכים במערכת, והמעבר למצב עם יותר תהליכים עולה פרופורציונית ל-λ. חזרה למצב הקודם עולה פרופורציונית ל-μ.



איור 32 : המחשה לשרשרת מרקוב

אחת התכונות של שרשראות מרקוב היא שיש להן התפלגויות גבולויות, שנסמנן ב-π_i, כלומר ניתן להסתכל על ההתפלגות בהיבט אסימפטוטי.

נאפיין את השרשרת באמצעות הנחת האיזון: $\pi_i \cdot \lambda = \pi_{i+1} \cdot \mu$, כלומר, אם התקדמנו קדימה, חייב להיות גורם שמאזן אותנו, ולכן ההסתברות לחזרה מהמצב הבא שווה. דבר זה נובע מכך שאם המערכת עובדת רק ב-20%, אז ב-80% היא לא עושה דבר, ולכן המעבר למצב עבודה ב-30%, מקביל למנוחה ב-70% מהזמן. כלומר ההבדל הוא 10%, אותו גודל.

מקשר זה אנו מקבלים כי

$$\begin{aligned}
&\pi_i = \left(\frac{\lambda}{\mu}\right)^i \pi_0 = \rho^i \pi_0 \\
&\Rightarrow 1 = \sum_{i=0}^{\infty} \pi_i = \pi_0 \sum_{i=0}^{\infty} \rho^i = \frac{\pi_0}{1-\rho} \\
&\Rightarrow \pi_i = \rho^i (1-\rho), \quad \rho = \frac{\lambda}{\mu}
\end{aligned}$$

מכאן מתקיים כי

$$\overline{n} = \sum_{i=0}^{\infty} i \cdot \pi_i = \sum_{i=0}^{\infty} i (1-\rho) \rho^i = \frac{1-\rho}{(1-\rho)^2} \rho = \frac{\rho}{1-\rho}$$

ממשפט Little מתקבל כי

$$\overline{r} = \frac{\overline{n}}{\lambda} = \frac{\rho}{\lambda (1-\rho)} = \frac{\frac{1}{\mu}}{1 - \frac{\lambda}{\mu}}$$

וקיבלנו בדיוק את הזמן הממוצע לתגובה. מקשר זה אנו מקבלים שכאשר $\frac{\lambda}{\mu}$ קרוב ל-1 כלומר יש ניצול מירבי, יש שאיפה של זמן התגובה לאינסוף, זה הגיוני, היות שיש המון עבודות שצריך לטפל בהן.

הערה. כל זה נכון בהנחה שהמודל של λ, μ נכון. כיוון שהמודל עצמו אינו דטרמינסטי, אלא הסתברותי, אנו מקבלים את השאיפה הזו לאינסוף ולכן במקרה ההסתברותי 90% ניצול או אפילו 50% יביא לקריסה. במקרה הדטרמניסטי שהכל קבוע, יתכן בהחלט ש-90% יעבוד טוב.

5.5.2 סוגים שונים של מערכות

מערכת Closed איננה פתוחה לעבודות חיצוניות ובלתי צפויות. הכל מוגדר, ידוע, ודטרמיניסטי. לכן הניצול הוא 100%. במערכת זו הקצב שעבודות מגיעות תלוי בקצב ה-Feedback כלומר בקצב סיום העבודות.

מערכת Open פתוחה לעבודות חיצוניות ובלתי צפויות. למשל לפתע התבקשנו לחשב משהו מוזר. זה אפשרי. או, שרת אינטרנט צריך לתת מענה לשאילתה על אתר שלא השתמשו בו כבר יום שלם. לכן הניצול צריך להיות קטן מ-100%, אחרת היא תקרוס ממש בקלות. למערכות מסוג זה יש התכונה שהבקשות מגיעות באופן בלתי תלוי בעומס המערכת.

מערכת Combined מערכת שמשלבת בין השתיים. למשל, סגורה, אבל עם אפשרות להשתמש בטרמינל. ניתוח מערכות כאלה הוא מורכב יותר.

5.5.3 צוואר בקבוק

יתכן שהעומס הוא בכלל לא המעבד על ה-I/O. למשל עם אנחנו מעבדים וידאו וכל פריים משנה ריבוע במסך לצבע שחור, נדרש שימוש מאסיבי בתקשורת עם המסך. לכן, התזמון על המעבד לא בהכרח ישנה את התוצאות משמעותית. במילים אחרות, עלינו לתזמן את הרכיבים העמוסים.

5.6 סוגים שונים של Scheduling

5.6.1 Long − Term Scheduling

עד כה הנחנו שהרצה של תהליכים תלויה אך ורק במעבד וברכיבי ה-I/O. אך יתכנו מקרים בהם אין לנו בכלל מספיק זכרון כדי להריץ את כל התהליכים במקביל. במצב זה צריך לתזמן אותם לטווח הרחוק. כלומר, עלינו להימנע מהרצה של חלק מהם, ולזכור זאת בהמשך. כמובן, תהליכים אינטרקאטיבים יקבלו עדיפות גבוהה ולא יזרקו (ככל הנראה), ועלינו ליצור מנעד מספיק מגוון של תהליכים שרצים. למשל, לא לאפשר רק לתהליכים אינטרקאטיבים לרוץ.

מהי אותה תערובת טובה? זו תערובת שתכלול עבודות מכל הסוגים - מערכת ההפעלה, אינטראקציה עם המשתמש, דיסק ועוד.

5.6.2 Fair − Share Scheduling

זהו תזמון שמסתמך על הדברים הבאים:

1. Virtual Time Scheduling - כל תהליך מקבל חלק יחסי (Fair, not necessarily equal) מהזמן של המעבד. למשל, במקום להעלות את ה-cpu_usage ב-1 בכל תיק של השעון, נעשה זאת בכל שלושה תיקים. ככה הוא ירוץ פי שלוש ממה שתהליכים אחרים ירוצו. במובן זה, הוא חווה זמן שונה. בכל פעם ירוץ התהליך עם היחס הקטן ביותר. שמירת תהליך היא פעולה יקרה ולכן צריך לבחור בקפידה את מי להעתיק.
2. Lottery Scheduling - במקום שהזמן יהיה שונה, כל תהליך מקבל מעין "כרטיסיה" של הזמן שהוא רשאי לרוץ בו או של משאבים אחרים. בכל פעם שהוא רץ או משתמש במשאב, מנקבים את הכרטיסיה.

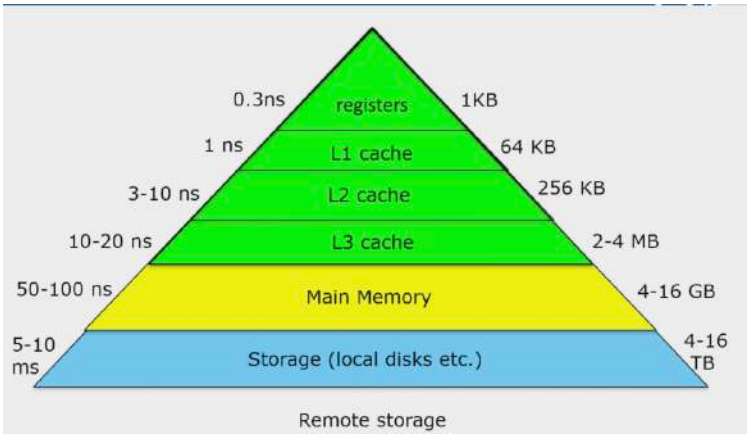
6 ניהול זכרון

6.1 מבוא

עד כה חילקנו את זכרון המחשב לשלושה חלקים:

1. אוגרים - כמות מזערית של זכרון שיש למעבד.
2. זכרון לטווח קצר - כמות זכרון שיש לתכנית במהלך ריצתה (RAM).
3. זכרון לטווח רחוק- דיסקים, המידע שנשמר שם לא נמחק אלא אם מחקנו אותו בעצמנו.

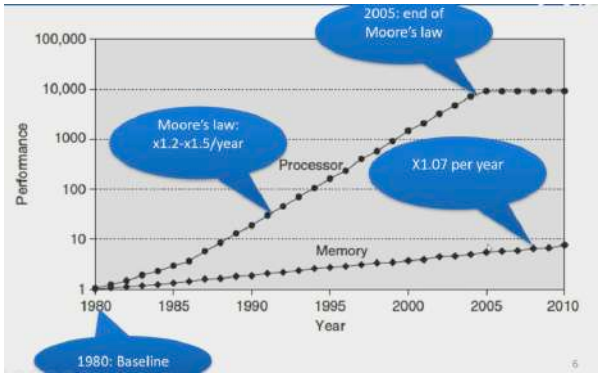
כמובן שכמות הזכרון גדולה יותר לפי סדר החלוקה שביצענו. עתה, נבצע חלוקה מפורט יותר של אזורי הזכרון:



איור 33: לשם השוואה, ns אחד הוא הזמן שלוקח לאור לעבור את המרחק בין קצה האצבע לקצה האגודל שלנו. זה זמן מאוד קטן. ככל שעולים בהיררכיה, ככה זמן הגישה עולה. למשל זמן הגישה לדיסק הוא בסדר גודל של 10ms שזה פי 10^7 בערך מ-ns, גישה לאוגרים. אם ns אנלוגי לכמה שניות, כלומר ללכת לקצה החדר ולהביא חפץ קטן, 10ms שקול להזמנת משלוח מאמזון.
מבחינת גודל הזכרון, הוא גם עולה כשעולים בהיררכיה. הזכרון הזמני קטן מעמותית מהזכרון לטווח הרחוק.

אנו רואים כי יש פער בין מהירות הגישה של המעבד, למהירות הגישה לזכרון הכללי.

הגרף הבא ניתן לראות השוואה בין מהירות הגישה המעבד למהירות הזכרון:



איור 34: בשלב מסויים הפסיקו להמשיך להשתמש בחוק מור, כיוון שהגדלת מהירות המעבד תגרום לו להישרף.

בשנת 1990 נוצר פער גדול בין מהירות המעבד למהירות הזכרון, דבר שגרם ליצור עזרים לגישה לזכרון, כדי למנוע עיכוב בפעולות המעבד.

עזרים אלה, הם זכרון ה-Cache, שהוא זכרון לגישה מהירה מהמעבד, בו נשמרים בלוקי מהזכרון שניגשנו אליהם בעבר, ויתכן מאוד שניגש אליהם בעתיד הקרוב. מציאת זכרון ב-Cache, ניהול הזכרון בו, מחיקה והוספה, זו אחריות של מערכת ההפעלה, והיא חלק מאלגוריתם ניהול הזכרון, ונלמד על כך במפורט בהמשך.

6.1.1 עקרון המקומיות (Priciple of Locality)

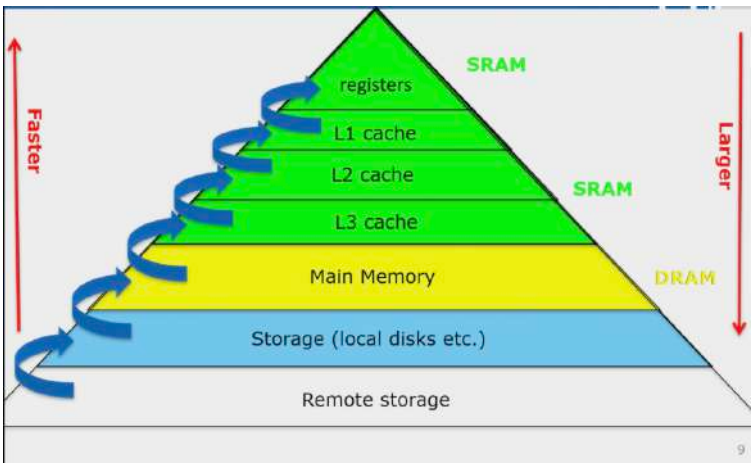
עקרון זה נוגע לשני רכיבים, זמן ומרחב:

- זמניות מקומיתי (Temporal Locality): אם ניגשנו למקום מסוים בזכרון, בסיכוי די גבוה, ניגש אליו שוב בקרוב. למשל, אם זה משתנה שאנחנו רוצים לעדכן אותו, או אם זה משתנה של תנאי בלולאה (i).
- מרחב מקומי (Spatial Locality): אם ניגשנו לכתובת מסויימת בזכרון, בסיכוי די גבוה, ניגש גם לשכנים שלה. למשל, במערכים, או בהרצה סדרתית, כלומר, אם אנו מריצים תכנית פקודה אחר פקודה, אז כדי להעתיק בלוק של פקודות בזכרון בכל פעם, ולא פקודה אחר פקודה (Sequential Instruction).

השימוש בעקרון מתבסס על כך שצריך לשמור את הזכרון קרוב למעבד.

6.1.2 RAM

למעשה, כל הגישה לזכרון מבוססת על רעיון ה-Caching:



איור 35: כאשר אנו ניגשים לזכרון מהענן או מעבירים אותו לדיסק המקומי. כשאנחנו פותחים אותו אנו מעבירים אותו לזכרון של התכנית שפתחה אותו, לאחר מכן, שלב שלב, מעבירים אותו למקומות שונות בזכרון ה-Cache (L1, L2, L3) עד שבסוף הוא מגיע לאוגרים והמעבד יכול לגשת אליו.

כדי לבצע את ההעברה אנו משתמשים בטכנולוגיות שונות. הטכנולוגיה המהירה ביותר היא SRAM אחריה DRAM ואחריה ה-DISK.
בגדול, RAM = Random Access Memory ושני הרכיבים שתיארנו הם:

- (SRAM) Static RAM: טכנולוגיה מהירה ויקרה יותר. לא רגיש לרעש אלקטרוני, משתמשים בו לאוגרים ול-Cache.
- (DRAM) Dynamic RAM: טכנולוגיה איטית וזולה יותר. מאוד רגיש להפרעות, משתמשים בו לזכרון הראשי.
- (DISK): הדיסק לעומת זאת, הוא פלאטה פיזית שמסתובבת, ויש זרוע מגנטית שקוראת ממנה מידע. הוא איטי משמעותית יותר מה-DRAM.

הגדרה. זכרון נדיף (Volatile) זכרון שנמחק כאשר אנחנו מכבים את המחשב.

הגדרה. זכרון לא נדיף (Non Volatile) זכרון שלא נמחק לאחר כיבוי המחשב.

6.1.3 אחריות על הזכרון

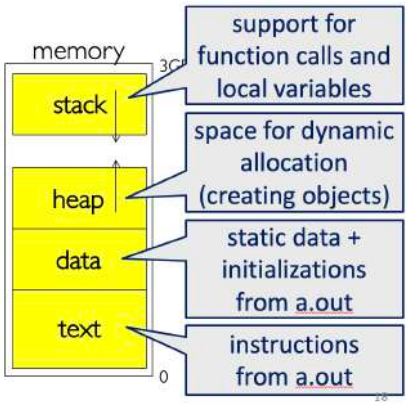
כפי שראינו, ניתן לתכנת MultiCore כך שכמה מעבדים ירוצו במקביל. אם כך, איזה זכרון משותף למעבדים ואיזה נשאר פרטי?
כל הזכרון עד ה-Cache L2 כולל, הוא פרטי לכל מעבד, כלומר מועתק. משם והלאה הוא משותף, זאת כיוון שאנחנו רוצים לאפשר לכל מעבד לשמור זכרון ב-Cache משלו, בלי להגן עליו, אבל גם זכרון משותף מהיר, מלבד הזכרון הפיסי והכללי.
עתה, נרצה לדעת, מי אחראי על כל אחד מחלקי הזכרון השונים. החלוקה מתבצעת, (איך לא?) כך:

- הקומפיילר אחראי על האוגרים.
- החומרה אחראית על ה-Cache (L1, L2, L3).
- מערכת ההפעלה אחראית על ה-Main Memory, Remote Storage. אחריות זו כוללת ניהול זכרון, החלפה ומערכת הקבצים.

6.2 מרחב הכתובות (Address Space)

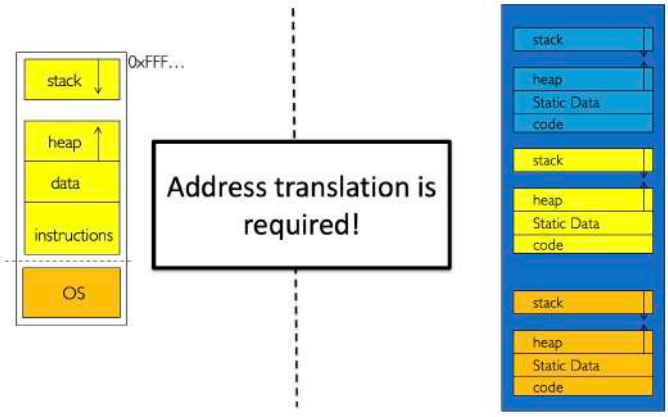
6.2.1 מרחב כתובות מקומי וגלובלי

מרחב הכתובות הוא קבוצת כל הכתובות שתהליך יכול לגשת אליו. הוא תלוי כמובן בארכיטקטורה.
בארכיטקטורת 32bit נקבל 4GB כתובות. חלק מהזכרון מושאר למערכת ההפעלה.
בפועל, ראינו כי מרחב הכתובות הוא החלוקה הבאה, וכי תהליך חושב שזו חלוקה של כל הזכרון:



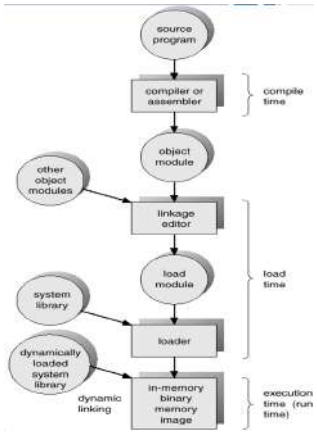
איור 36: חלוקה של מרחב הכתובות והתפיקידם של כל חלק.

למרות זאת, בפועל, כמה תהליכים רצים במקביל, או ב-Time Sharing, או שיש כמה חוטים בתהליך עם Stack שונה, כך שמתקבל המצב הבא:



איור 37: מרחבי כתובות מרובים עבור תהליכים שונים, אל מול נקודת המבט של התהליך בה הוא רואה מרחב כתובות יחיד.

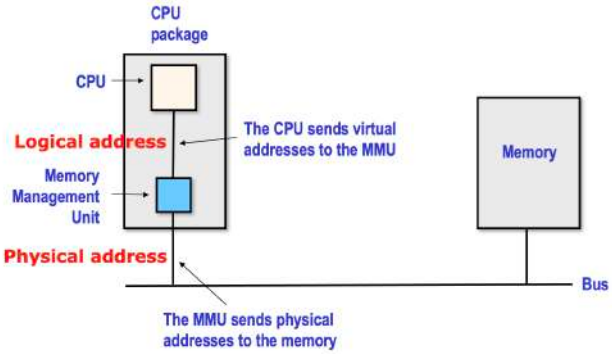
כדי לאפשר מצב זה, בכל גישה של התהליך לזכרון, הוא ישלח כתובת שתעבור דרך AddressTranslator, שיהפוך אותה לכתובת במרחב המלא.
למשל, נוכל לשמור פוינטר base לתחילת הזכרון שהתהליך רואה, ובכל גישה לזכרון x ניגש בפועל זכרון base + x.
בעבר, חילקנו את שלבי הרצת התכנית לשלב קומפילציה, Linkage ו-Load שמתבצע ברכיב ה-Loader. רכיב זה, בין השאר, ממיר את הכתובות שהתהליך רואה לכתובות אמיתיות בזכרון של המחשב.



איור 38 : דיאגרמה לתהליך ה-Load בעת הרצת תכנית

למעשה, רובן המוחלט של הכתובות ידועות אך ורק לאחר שלב ה-Load. בפועל, כיצד נממש Loader? הרי אנו יודעים שלאחר ה-Linkage יש קובץ המוכן להרצה.

הזכרון שלנו יכלול יחידה לטיפול בזכרון (Memory Unit).



איור 39 : כאשר המעבד רואה כתובת בזכרון היא עוברת דרך ה-Memory Unit ובכך הגישה היא לזכרון "האמיתי" ולא לכתובת שהתהליך רואה.

6.2.2 סגמנטציה

הפתרון שנתנו לביצוע ההמרה בין הכתובות, איננו פרקטי. שכן אינו לנו מספיק זכרון פיסי להכיל את כל הזכרון של התהליכים הרצים, ולכן לא ניתן לבצע את המיפוי הנ"ל. הדרך לעשות זאת היא על ידי שימוש בסגמנטציה.

נזכור כי מרחב הכתובות כולל בתוכו את כל הכתובות שאפשר לייצר במערכת 32Bit. בפועל, התכנית שלה קטנה יותר משמעותית ממרחב כל הכתובות, ולכן, עדיף למקם רק חלק ממנו. במקרה שלנו, נחלק את מרחב הכתובות לסגמנטים בגודל מוגבל, ולכל אחד נשמור פוינטר Base כפי שתיארנו קודם. במקרה הזה, אנו לא ממפים את כל מרחב הכתובות אלא רק ממנו, שכן גדלי הסגמנטים מוגבלים. כמו כן, כל סגמנט הוא בלוק רציף בזכרון, בעוד סגמנטים שונים לא חייבים להיות אחד אחרי השני בצורה רציפה. עם שינוי זה, אנו מסוגלים למפות את הסגמנטים של כל התהליכים.

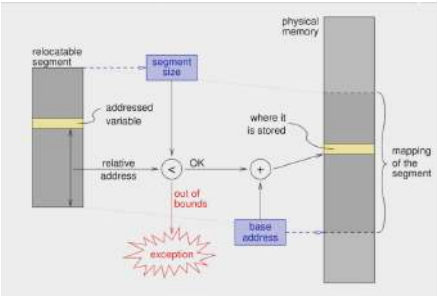
בשלב זה, הגיע הזמן לתת שם לכתובת שהתהליך רואה, אל מול הכתובת האמיתית:

הגדרה. כתובת לוגית היא כתובת זכרון שתהליך רואה בזמן ריצתו, ואיננה בהכרח הכתובת אליה צריך לגשת בזכרון הפיסי. כלומר, בעת הגישה, צריך לבצע המרה. באלגוריתם שהצענו, מדובר ב-x.

הגדרה. כתובת פיסית היא הכתובת אליה צריך לגשת בזכרון הפיסי כדי למשוך את המידע מהזכרון. באלגוריתם שהצענו מדובר ב-Base + x.

יחד עם זאת, עלינו לוודא שכתובת לא גולשת מחוץ לסגמנט שהוגדר. למשל, אם גודל הסגמנט הוא 1000 בתים, וביקשנו לגשת לכתובת ה-1001, עבורנו, זו לא כתובת חוקית, ולכן עלינו לוודא שהדבר לא אפשרי.

סך הכל, אנו מקבלים את הדיאגרמה הבאה:



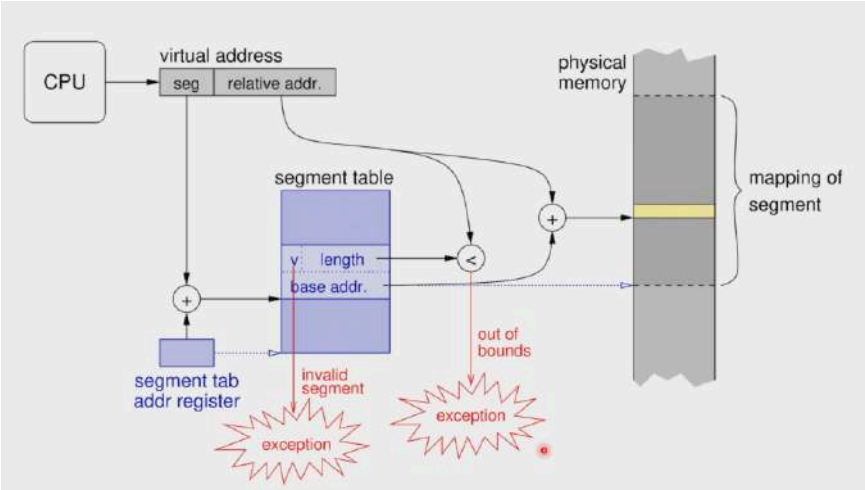
איור 40: המחשה לתהליך מיפוי הכתובות. כל עוד הכתובת לא גולשת מגודל הסגמנט, נמיר אותה לכתובת פיסית. אחרת, המעבד יזרוק Exception.

6.2.3 Segment Table Address Translation

לכל תהליך יהיו הרבה סגמנטים שונים, ולכל אחד צריך לשמור את ה-Base, Bound כדי שיהיה אפשר לבדוק חוקיות של כתובת, ולהמיר לכתובת פיסית. דבר זה דורש יצירת טבלת מידע עבור הסגמנטים שתכיל את כל ערכים אלה. לטבלה זו קוראים Segment Table.

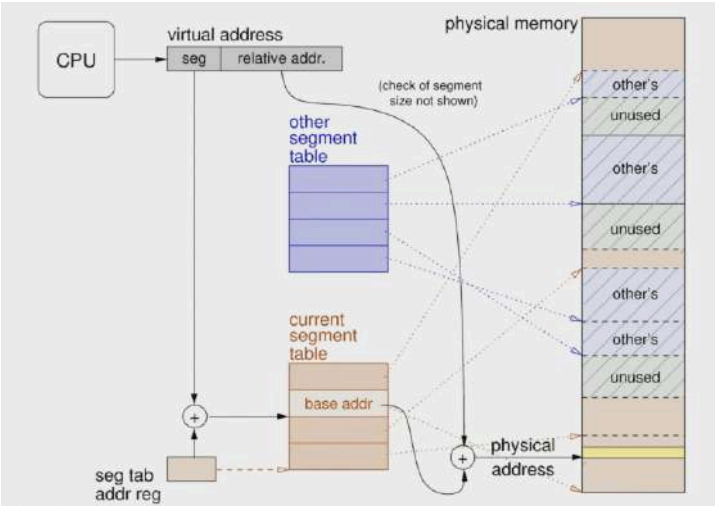
כל כתובת בתהליך תכיל סגמנט וכתובת לוגית, כלומר $\boxed{\text{Seg} \mid \text{Relative Addr}}$. כדי לתרגם אותה, ראשית נבדוק האם מדובר בסגמנט חוקי, כלומר האם הוא חוקי לגישה, את זאת אנו בודקים על ידי הסתכלות על ה-Validation Bit שנסמנו V. אם לא, נזרוק חריגה (המעבד יזרוק).

אם הוא חוקי, נשלוף מטבלת הסגמנטים של התהליך את אורך הסגמנט ואת ה-Base, כלומר ניגש לטבלה במקום ה-Seg. לאחר מכן נשווה את הכתובת הרלטיבית לאורך הסגמנט, ואם היא חורגת ממנו, נזרוק חריגה. סך הכל אנו מקבלים את הדיאגרמה הבאה:



איור 41: המחשה לשימוש בטבלת הסגמנטים. **יש שתי שגיאות, סגמנט לא חוקי** (Invalid V)**, או חריגה מהסגמנט** (Seg Fault)**.**

כאשר מתבצע Context Switch למעבר לתהליך אחר, צריך לספק למעבד טבלת סגמנטים חדשה. כיצד נעשה זאת? נחליף את הכתובת השמורה ברגיסטר טבלת הסגמנטים לכתובת של טבלת הסגמנטים החדשה. במילים אחרות, יש רגיסטר מיועד לשמירת הכתובת של הטבלה. זה נראה כך:



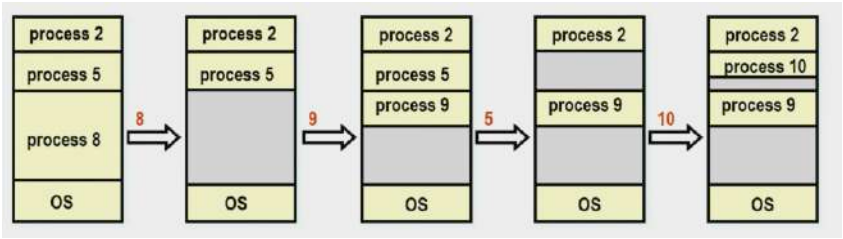
איור 42: ניתן לראות כיצד רג'יסטר הכתובת של טבלת הסגמנטים מכיל כתובת המצביעה על הטבלה. כמו כן, יש מספר טבלות שונות של תהליכים שונים, וכאשר נבצע Context Switch, נגרום לרג'יסטר להצביע לטבלה אחרת.

6.3 מערכת ההפעלה - Allocation Dynamics

בתהליך הסגמנטציה, איפה מערכת ההפעלה פועלת? החלפת הכתובת ברג'יסטר היא אוטומטית, אך ביצוע ה-Context Switch הוא אחריות של מערכת ההפעלה. כמו כן, הקצאת סגמנטים לכל תהליך, וזכרון לתהליך, הן אחריות של מערכת ההפעלה גם כן.

6.3.1 Fragmentation

יתכן שנשחרר תהליך שהזכרון שלו הוא בין שני תהליכים קיימים, מה שייצור מרווח בין התהליכים:



איור 43: בשלב מסויים תהליך 9 מפריד בין שני בלוקים פנויים בזכרון, מה שמונע מאיתנו להקצות זכרון לתהליך שנכנס רק בסכום הבלוקים.

למצב המתואר באיור קוראים External Fragmentation.

יש מצבים בהם אנו מקצים לתהליך יותר זכרון ממה שהוא מבקש, על מנת שבמידה שהוא ירצה לגדול, הוא יוכל. במצב זה יש זכרון פנימי בתהליך שלא בשימוש, ולו אנו קוראים Internal Fragmentation.

אנו נרצה לטפל ב-External Fragmentation, אשר מהווה Overhead. Internal Fragmentation יצרנו במכוון.

על כן, אחד השיקולים בניהול זכרון הוא טיפול ב-Fragmentation.

6.3.2 Segmentation Algorithms

כדי להקצות זכרון לתהליך, נרצה לוודא שהפרגמנטציה היא מינימלית. אך לפני כן, נציג אלגוריתמים שונים לביצוע הפעולה. כל האלגוריתם יקבלו את אותו הקלט ויתנו את אותו הפלט.

קלט רשימה של אזורים משוחררים בזכרון ובקשה לבלוק בזכרון.

פלט החלטה באיזה בלוק משוחרר להשתמש.

First Fit אפשרות אחת היא להחזיר את הבלוק הראשון שמתאים. זה לינארי עם קבוע < 1. הבעיה היא שזה עלול ליצור פרגמנטציה גדולה מדי. נעיר כי במקרה זה, נרצה שהתור יהיה ממוין לפי הכתובות, שכן על מנת לתקן פרגמנטציה, נרצה שיהיה יחס הגיוני בין האזורים המשוחררים. יחד עם זאת, זה ייצור פרגמנטציה גדולה יותר בכתובת נמוכות, שכן בוחרים את הכתובת הראשונה שמתאימה והתור ממויין.

Next Fit נסרוק את התור, מהמקום שעצרנו פעם קודמת, ונבחר את הבלוק הראשון שמתאים. גם כאן זה לינארי עם קבוע < 1, ונקבל פרגמנטציה שמחולקת באופן שווה יותר, שכן עוברים על כל התור, ולא רק על כתובות המופיעות בתחילתו.

Best Fit נעבור על הרשימה ונבחר את הבלוק שהגודל שלו הכי קרוב לגודל הבלוק שהתבקשנו למצוא. זה יצמצם מאוד פרגמנטציה, שכן לוקחים זכרון כגודל הבלוק, אותו במילא צריך לקחת. במקרה זה החיפוש הוא לינארי, הדורש מעבר על כל התור, והתוצאה משמרת בלוקים גדולים, ומקטינה את הפרגמנטציה.

6.3.3 Fragmentation Solutions

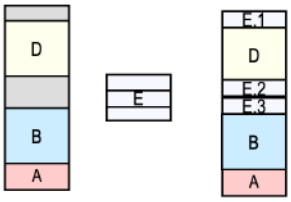
פתרון אפשרי לבעיית הפרגמנטציה הוא הצמדה של בלוקי זכרון אחד לשני. למשל, אם יש שלושה בלוקים A, B, C הצמודים אחד לשני באופן רציף, אך B הוא בלוק משוחרר, נעתיק את C צמוד ל-A כך שנקבל A, C.

הבעיה בפעולה זו היא המחיר שלה, שכן העתקה של תהליך שלם ממקום למקום זו פעולה יקרה מאוד.

6.4 Paging

על מנת לטפל בפרגמנטציה באופן יעיל, נשתמש ב-Pages. במקום להעתיק את כל התהליך ממקום למקום, מלכתחילה, נקצה את התהליך באופן לא רציף. כלומר, נחלק את התהליך ליחידות קטנות יותר, ונשים כל אחת במיקום אחר בזכרון. כך נוכל להתגבר על פרגמנטציה, ולנצל מרווחים בזכרון כדי לאכסן חלקים מתהליך.

לכל יחידת זכרון זו, אנו קוראים Page.



איור 44: במקום להצמיד את D ל-B. חילקנו את E ל-Pages ומיקמנו כל Page במיקום מתאים בזכרון, בין השאר במרווחי הפרגמנטציה.

תהליך יצירת ה-Paging מתבצע על ידי הסתכלות מעט שונה על הזכרון. במקום להסתכל על הזכרון כרצף של בתים, נסתכל עליו כרצף של Frames כאשר הגודל של כל Frame קבוע. נגדיר את הגודל של כל Page להיות הגודל של Frame ונקבל שאם נתאים Page ל-Frame תתקבל אפס פרגמנטציה.

מכאן, מטרת מערכת ההפעלה היא למפות כל Page של תהליך ל-Frame בזכרון. יחידת הזכרון במעבד תדאג למפות את הכתובות בהתאם.

לכל תהליך תהיה טבלה בה לכל Page יתאים ה-Frame בזכרון הפיסי. היות שה-Pages נשמרים בצורה מסודרת בטבלה וממוספרים מ-0 בסדר עולה, מספרם הוא בדיוק האינדקס המתאים בטבלה, ולכן מספיק לשמור מערך עם מספרי ה-Frames.

6.4.1 Address Translation

הכתובת הוירטואלית שתהליך ישתמש בה תורכב מ-32 ביטים (או 64 אם זו מערכת 64 ביט) כך ש-

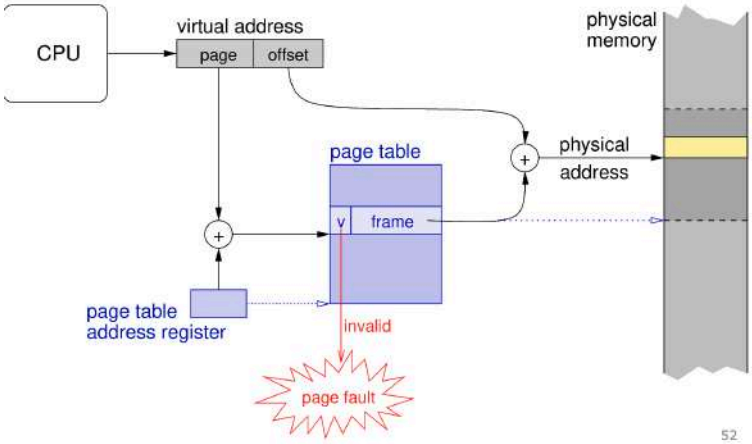
- 20 הביטים העליונים ייצגו את מספר ה-Page.
- 12 הביטים התחתונים ייצגו את ההיסט בתוך ה-Page.

נשתמש במספר העמוד כאינדקס בתוך טבלת ה-Pages ונקבל את ה-Frame באמצעות הטבלה, שיהיה בגודל 20 ביטים. נשלב את כתובת ה-Frame יחד עם ההיסט ונקבל את הכתובת הפיסית.

כמו במקרה של סגמנטציה, למעבד יהיה רג'יסטר שישמור את הכתובת של טבלת העמודים של התהליך הנוכחי.

כאשר תתקבל כתובת [Pages, Offset] ניגש לטבלה במקום ה-Pages, ונבדוק את ביט החוקיות. אם זה חוקי, נמשיך הלאה, אחרת נזרוק Exception. לאחר מכן, נשלוף את כתובת ה-Frame ונחזיר [Frame, Offset] ככתובת הפיסית.

לחריגה שתיארנו קוראים Page Fault.



איור 45: המחשה לתרגום הכתובת הוירטואלית לכתובת פיסית.

הערה. בעת חישוב הכתובת הפיסית, איננו מבצעים פעולות חיבור, אלא פעולת הדבקה שממומשת ברמת החומרה. שכן על אף שמדובר כביכול במספר מאוד גדול, הוא מכיל המון אפסים בביטים ההתחלתיים שלו, שעליהם אנו מדביקים את ההיסט. על כן אנו חוסכים המון CPU Cycles.

6.4.2 Overheads

שימוש ב-Pages יוצר סוגים שונים של Overheads:

- אכסון הטבלה דורש לא מעט מקום בזכרון. הדרך להתמודד עם זה הוא לבצע אופטימזציה לגודל ה-Page, או להשתמש בטבלות דפים מורכבות יותר.
- גישה לטבלה דורשת גישה נוספת לזכרון, שזה דבר איטי מאוד. הפתרון הוא לשמור Cache יעודי לטבלות עם מבנה שנקרא "TLB".

6.4.3 גודל דף אופטימלי

בקביעת גודל הדף יש לנו טרייד אוף.

כאשר אנו ממפים את הדף האחרון של סגמנט, יתכן שהוא יכיל עוד זכרון נוסף שנובע מכך שהסמגנט לא מורכב ממספר שלם של Pages, ולכן, אם נקבע דפים קטנים, נקבל פרגמנטציה פנימית קטנה יותר. הבעיה היא שאם נקטין את גודל הדף, נקבל טבלה גדולה יותר. בממוצע, נשער שיתקבל $\frac{1}{2}$ מגודל הדף, שכן יש מקרים בהם כמעט כל הדף הוא פרגמנטציה, חלק מזערי הוא פרגמנטציה, ובאמצע נקבל בדיוק $\frac{1}{2}$.

נסמן:

- p - גודל הדף.
- s - גודל התהליך.
- e - גודל הכניסה בטבלת הדפים, כלומר גודל ה-Frame, והביט שמראה אם הדף ממופה או לא.

ה-Overhead מורכב משני חלקים:

1. השטח הדרוש לאחסון הטבלה: יש לנו $\frac{s}{p}$ כניסות, כל אחת בגודל e ולכן $\frac{s \cdot e}{p}$.
2. הפרגמנטציה הפנימית הממוצעת: $\frac{p}{2}$.

על כן:

$$\text{Overhead} = \frac{s \cdot e}{p} + \frac{p}{2}$$

על מנת לקבל גודל דף אופטימלי, נגזור לפי p ונשווה לאפס:

$$\frac{d\,(\text{Overhead})}{dp} = \frac{1}{2} - \frac{s \cdot e}{p^2} = 0$$

מכאן נקבל כי

$$\boxed{p_{\text{opt}} = \sqrt{2 \cdot s \cdot e}}$$

דוגמה. עבור $s = 1\text{MB}, e = 64\text{Bit}$ מתקבל כי $p_{\text{opt}} = 4\text{KB}$.

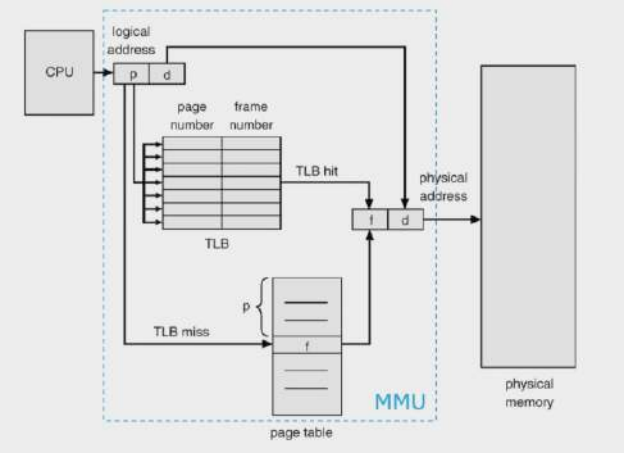
6.4.4 תרגום כתובות עם TLB

ניזכר כי בתהליך תרגום הכתובת הלוגית לכתובת הפיסית אנו מבצעים את ההליך הבא:

```
void address_translation(int page, int offset){
        int frame_bits = page_table[page];
        if (!(frame_bits & 1)){ // page is not mapped, first bit is zero
                throw Exception;
        }
        else if (offset >= ((frame_bits << 1) & LENGTH_MASK){ // offset is bigger than
            frame length
                throw Exception;
        }
        return frame_bits << BASE_ADDR_BITS; // extract the base addr from the frame
            bits
}
```

נבחין כי אנו מבצעים גישה אחת לזכרון על מנת לקבל את הכניסה המתאימה ל-Page בטבלה. גישה לזכרון זה דבר יקר, ולכן היינו רוצים להימנע ממנה. על כן, נשתמש ב-Cache ייעודי עבור הטבלה, שיכיל מבנה נתונים שנקרא לו ה-TLB = Translation Lookaside Buffer.

הוא ימפה את מספר ה-Page לערך ה-Frame בטבלה, וכך יחסוך את הגישה האיטית. עם תוספת זו, נקבל את התרשים הבא:



איור 46: המחשה לפעולת תרגום הזכרון הלוגי לפיסי באמצעות ה-TLB

היות שגודל ה-TLB מוגבל, יתכנו שני מקרים -

1. מצאנו את ה-Page ב-TLB - מצב זה יקרא TLB hit, ובוא קיבלנו את מבוקשנו.
2. לא מצאנו אותו ב-TLB - מצב זה יקרא TLB miss, ואנו נאלץ לגשת לזכרון ולחפש בטבלה.

הערה. ה-TLB הוא בדרך כלל Fully Associative, כלומר מאפשר לקשר בין מספר דף לכל Frame, ואף מאפשר חיפוש מקבילי בתוכו.

גודל ה-TLB הוא בדרך כלל קטן - 64 כניסות, שזה אומר $64 \cdot C$ כאשר C גודל כל כניסה. הוא ממומש בטכנולוגית SRAM, שכפי שציינו בעבר, היא מאוד מהירה.

6.5 Virtual Memory

המהות של הזכרון הוירטואלי מתבטאת בשאלה הבאה:

שאלה מה יש יותר, כתובות לוגיות, או כתובות פיסיות?

נבחין כי כאשר תהליך רץ, הוא מבודד, וחושב שהוא יכול לגשת לכל זכרון התכנית, ואכן, תאורטית הוא היה יכול לעשות זאת. לכן, יש לכל הפחות כתובות לוגיות כפיסיות.

יוצא איפה, שכל תהליך חושב שהוא יכול לגשת לכל הזכרון הפיסי, ואנו מיישבים את הסתירה הזו על ידי הקצאה מראש של זכרון עבור התהליך וביצוע ואלידציה בעת תרגום הכתובת הלוגית לכתובת הפיסית.

יחד עם זאת, נבחין כי **תכנית לא משתמשות בכל מרחב הכתובות שלהן כל הזמן,** אלא רק בחלקים בכל פעם.

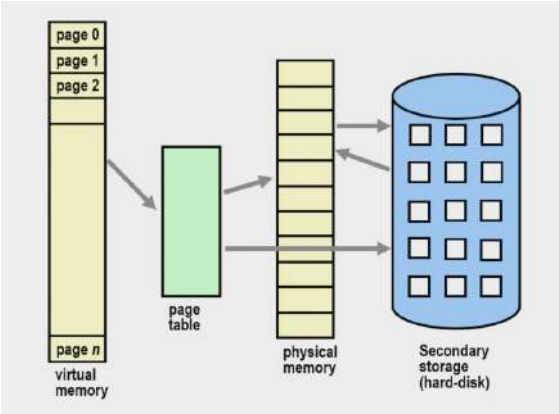
למשל, לאחר שהשתמשנו בקוד אתחול, לא צריך לשמור אותו. אם הקוד תקין, ואין צורך בקוד להתמודדות עם שגיאות, אין צורך לשמור אותו בזכרון של התהליך.

על כן, נסיק כי חלקים שלא משתמשים בהם, לא צריכים להיות ממופים לזכרון התכנית. אחת העובדות החשובות, היא ששמירת זכרון על הדיסק, זולה בהרבה משמירה על ה-RAM, ולכן נוכל לשמור את המידע שלא בשימוש בדיסק. דבר זה יוריד את כמות הזכרון שנצרכת על ידי התכנית. יתר על כן, נוכל לטעון תהליכים באופן יעיל יותר, שכן נצטרך לטעון אך ורק את החלקים החיוניים בהווה, ולא אלא שהיו חיוניים. למשל, נוכל לטעון תהליך עם זכרון אפס, ולטעון את חלקי הזכרון הרלוונטיים להרצתו רק לאחר מכן.

Virtualization ניתקנו את התהליך ממרחב הכתובות הפיסי, וחיברנו אותו למרחב כתובות לוגי, מה שמנתק אותו גם מהגבלת הזכרון שיש לנו. זה גורם לו לחשוב כאילו שיש לו את כל הזכרון.

6.5.1 Dynamics of Demand Paging

המימוש של מה שהצענו, דורש שימוש ב-Paging, שכן בכל שלב, אנו או טוענים את הדף מהדיסק לשימוש עתידי, או קוראים אותו מהזכרון. כלומר, יש דפים שממופים בטבלה לזכרון הפיסי (RAM), ויש דפים שממופים לזכרון הדיסק:



איור 47: המחשה למיפוי הדפים - או לזכרון ה-RAM, או לדיסק.

למעשה, כאשר אנו ניגש לטבלת הדפים, יש שתי אפשרויות, או שנקבל $v = 1$ כלומר הדף בזכרון הפיסי (Main Memory), או ש-$v = 0$ ואז הדף הוא בזכרון הדיסק. במקרה הזה, אנו נעביר אותו מהדיסק ל-Main Memory.

הערה. דרך נוספת לביצוע הטעינה של דפים רלוונטים, היא Prepaging, שמסתמכת בעקרון הלוקליות. בשיטה זו, מערכת ההפעלה תנחש **מראש** אילו דפים התכנית תצטרך, וטוענת אותם מראש לזכרון. דבר זה חוסך הרבה זמן אם מערכת ההפעלה מנחשת נכון, אך בעל הרבה Overhead במידה שהיא טועה.

6.5.2 Page Fault

שאלה מה קורה כאשר הדף שמנסים לגשת אליו אינו נמצא בזכרון? (לא בזכרון הפיסי, אבל כן בדיסק).

במקרה זה, המעבד מקבל כתובת וירטואלית, אך בניסיון לתרגם אותה לכתובת פיסית, הוא מגלה שהיא לא ממופה לכתובת פיסית. לכן, זה מצב של Unmapped page.

כדי להתריע על כך למערכת ההפעלה, יחידת הזכרון (MMU) תיצור חריגה מסוג "PAGE FAULT", כלומר תיגש ל-Interrupt Vector במקום המתאים ל-Page Fault שם יש כתובת לפונקציה ייעודית של מערכת ההפעלה שתטפל בזה. הפונקציה תרוץ ותטען את קוד הטיפול ב-Interrupt. הבעיה היא שצריך לטעון את הזכרון של פונקציה זו ל-RAM, וכפי שתיארנו קודם, הוא ככל הנראה בדיסק. לכן מערכת ההפעלה מרדימה את התהליך עד שמסתיים תהליך הקריאה של הקוד ל-RAM, שזה סדר גודל של שבועיים ביחס למעבד.

בזמן זה מערכת ההפעלה עושה Context Switch ומריצה תהליך אחר. לאחר מכן, כאשר המידע של הפונקציה מגיע, מערכת ההפעלה מעירה את התהליך ושמה אותו ב-Ready Queue, כך שכשאר היא תריץ אותו, היא תריץ בדיוק את הפקודה הבאה.

למעשה, כאשר המעבד יראה שהדף לא חוקי, הוא ייצור Page Fault כפי שתיארנו. הפונקציה שמטפלת בו, תטען את הדף הרלוונטי מהדיסק אל הזכרון הפיסי, ותעדכן את הטבלה שמדובר בדף חוקי, כך שבהמשך יוכל התהליך לגשת אליו.

6.6 Replacement Algorithms

בעיה. מה נעשה אם אין Frame פנוי עבור הדף?

יש לנו רק דבר אחד סביר שנוכל לעשות והוא Page Eviction. רוצה לומר, נבחר דף שממופה ל-Frame, נעתיק את המידע שלו לדיסק, ונעתיק את המידע של ה-Page החדש ל-Frame. עלינו לבחור מי יהיה הקורבן שלנו.

בשלב זה נרצה לבנות אלגוריתם לבחירת הקורבן. אחת הדרכים היא לבחור דף ראנדומית, אך עדיף לא לבחור דפים שמשתמשים בהם הרבה, כי נצטרך לטעון אותם מחדש תוך זמן קצר.

יש הרבה אפשרויות שפתוחות עבורנו:

FIFO, Random, NRU (Not Recently used) , LRU (Least Recently Used)
Pseudo LRU, LFU (Least Frequently Used)

אלגוריתם אופטימלי (Infeasible) בכל שלב נבחר את העמוד שלא נשתמש בו הכי הרבה בעתיד. על פניו הכל טוב ויפה, הבעיה היא שאנחנו לא יודעים את העתיד. יחד עם זאת, אם נמצא אלגוריתם שמתקרב לאלגוריתם זה, נוכל לקבוע שהוא טוב.

אלגוריתם ראנדומי בכל שלב נגריל את הדף שנבחר. אנו יכולים לבצע זאת, אך אנו מתעלמים מכל המידע שיש לנו. אם האלגוריתם שנציע יהיה פחות טוב מראנדומי, נוכל לקבוע שהוא לא טוב.

שני האלגוריתמים הנ"ל יוצרים טווח בו כל האלגוריתמים בעלי ביצועים בטווח זה, קבילים, וכל מי שמחוצה לו, לא קביל.

אלגוריתם FIFO מערכת ההפעלה תבחר את מי להחליף בכל פעם, והיא תסתמך על מידע שיש לה. במקרה זה, היא תמיין את הדפים לפי הסדר שבו הם הגיעו לזכרון, ושומרת על סדר הרשימה, כך שכל פעם שדף חדש נטען, הוא נכנס לסוף התור. כאשר נרצה לבחור דף להחלפה, נוציא את הדף הראשון ברשימה. **הבעיה** באלגוריתם זה, היא שיתכן שהדף הראשון בתור הוא הדף שנמצא בשימוש הכי תדיר. זה לא הרבה יותר טוב מאלגוריתם אקראי, זה רק מונע החלפה עם דפים שהגיעו זה הרגע. למרות זאת, מערכת ההפעלה Windows NT משתמשת בשיטה זו, שכן זה אלגוריתם שלא דורש שום מידע על החומרה, ולכן הוא בלתי תלוי בחומרה.
חסרון נוסף הוא סתירה פנימית באלגוריתם. נניח שהגדלנו את כמות ה-Frames, האם יהיו יותר Page Faults או פחות? אנו מצפים שפחות, שכן יש יותר זכרון. למרות זאת, באלגוריתם זה יתכן שנקבל דווקא יותר.

דוגמה. נביט במקרה בו אנו ממפים את הדפים לפי הסדר הבא:

$$1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5$$

ויש לנו 3 Frames. נמפה תחילה, כאשר עמודה מסמנת את הדפים הממופים ברשימה, לפי הסדר (למטה זה הדף הראשון), התקדמות ימינה בעמודות מייצגת התקדמות בזמן:

1	2	3	4	1	2	5	5	3	4	4
	1	2	3	4	1	2	2	5	3	3
		1	2	3	4	1	1	2	5	5

יש מקרים בהם אין שינוי, כיוון שהדף כבר טעון. בסגול סימנו את ה-Page Faults. עתה, נניח כי יש 4 Frames. נקבל את המיפוי הבא:

1	2	3	4	4	4	5	1	2	3	4	5
	1	2	3	3	3	4	5	1	2	3	4
		1	2	2	2	3	4	5	1	2	3
			1	1	1	2	3	4	5	1	2

נבחין כי עבור 3 קיבלנו 9 Page Faults בעוד עבור 4 קיבלנו 11 Page Faults, ולכן קיבלנו יותר שגיאות עבור יותר זכרון. זה בעייתי.

עלינו להחליט מה אנחנו בדיוק רוצים מהאלגוריתם. ברור שאנחנו רוצים מזעור של מספר ה-Page Faults, אבל מה בדיוק אנחנו מחפשים?

אנו רוצים לטעון את הדפים שתהליך ישתמש בהם ברגע נתון, קבוצה שאנו מכנים ה-Working Set. עלינו לקבוע בדיוק מהי קבוצה זו.

נוכל להגדיר זו בצורה פרמטרית. עבור $k \in \mathbb{N}$, נקבע את ה-Working Set של תהליך להיות הדפים שהוא ניגש אליהם ב-k הגישות האחרונות לזכרון.
במקרה זה, כאשר הבחירה של הדפים ראנדומית, נקבל $|\text{Working Set}| \approx k$, ואילו אם הגישה היא לוקאלית, נקבל $|\text{Working Set}| << k$.

עלינו למצוא את k האופטימלי. תאורטית, נוכל להתחיל מ-$k = 1$, ולהגדיל אותו עד שה-Working Set ייתייצב. אבל זה לא מאוד פרקטי.

פרקטית, נוכל לקבוע את k להיות מספר מאוד גדול, נגיד 10^5, ונבחר את העמודים שהיו הכי הרבה בשימוש.

כדי שנוכל לעשות זאת, נצטרך תמיכה מהחומרה. היא תספק לכל דף עוד שני ביטים:

- Reference Bit (Used Bit) - ביט זה ידלק כאשר יגשו לדף.
- Dirty Bit - ביט זה ידלק כאשר משנים את הדף, כלומר ניגשים וכותבים אליו.

השינוי של הביטים הנ"ל מתבצע על ידי ה-MMU.

אלגוריתם ה-NRU (Not Recently Used) איננו יכולים לדעת את העתיד, אז ננסה לחזות אותו, לפי העבר. נשתמש בביט הרפרנס של הדפים, ובאופן מחזורי, ננקה אותם, כלומר בכל פסיקת שעון, נהפוך אותם להיות אפס. במהלך המחזור, יהיו תהליכים שינסו לגשת אליהם, ולכן ה-MMU יגדיר אותם להיות אחד. כאשר נרצה לבחור דף להחלפה, נבחר דף שיש לו רפרנס ביט עם ערך 0, באופן ראנדומי, שכן מאז הפעם האחרונה שניקינו את הביטים, אף אחד לא ביקש אותו. זה מאוד גס, אבל ניתן למימוש.

נוכל לשפר זאת באלגוריתם הבא:

אלגוריתם ה-LRU (Least Recently Used) היינו רוצים שהדפים היו מסודרים בשורה לפי מי שלא היה בשימוש הרבה זמן, ומי שנמצא בשימוש תדיר, כלומר ממש רשימה ממוינת לפי תדירויות. דבר זה נותן קירוב טוב ל-Working Set, ויש לו דיוק טוב יותר מה-NRU. הבעיה היא שהוא קשה למימוש, ויקר. למשל, נשמור בכל דף Time Stamp עם הזמן האחרון שהשתמשנו בו, ובכל החלפה של דף, נבחר את הדף עם ה-Time Stamp המינימלי - זה לינארי במספר הדפים, שזה יקר. דרך נוספת, הוא שימוש ברשימה מקושרת, כאשר נשמור בסוף הרשימה את הדף האחרון שנגענו בו, וככה נוכל לבצע החלפה מהירה, הבעיה היא שכדי לשחרר דף, נצטרך להגיע עד הזנב, ולכן זה עדיין לינארי במקרה הגרוע. אפשרות אחרת היא תחזוק של רשימה ממוינת ב-$n \log n$ מה שיוצר Overhead של $\frac{\log_2 n}{\text{Page}}$.

מסקנה. אפשר לממש LRU אבל זה יקר.

היות שה-LRU יקר, רוב מערכות ההפעלה לא משתמשות בו.

אלגוריתם ה-Clock נסתכל על הדפים כרשימה מעגלית, כאשר יש מצביע על אחד האיברים בה. לכל דף יש רפרנס ביט.
כאשר צריך לבצע החלפה, נסתכל על האיבר שמצביעים עליו ברשימה, אם הרפרנס ביט שלו הוא 0, נבצע איתו החלפה. אחרת, אם הוא 1, נהפוך את הביט ל-0 ונעבור הלאה. כלומר, ניתן לו הזדמנות שנייה. כלומר, אנו מחליפים את הדף הראשון שיש לו רפרנס ביט 0. זה קירוב ל-LRU.

יחד עם זאת, מדובר באלגוריתם גס. למשל, אם כל הביטים הם 1, כלומר יש הרבה עומס, הוא נהיה קרוב ל-FIFO ולכן קרוב לראנדומי. אבל כאשר יש זרימה, הוא סביר.

ניתן לבצע את השיפור הבא:

אלגוריתם ה-Clock Time במקום להסתכל אך ורק על ביט הרפרנס, נוכל לנסות לשמור את הזמן מהפעם האחרונה שנגענו בו. כדי לעשות זאת, אנו שומרים Time Stamps לכל דף. הבעיה היא שיותר מדי חתימות זמן, ידרשו מיון, ותחזוק יקר, לכן צריך שהזמן לא יהיה ברזולוציה של החומרה, כי היא תיצור יותר מדי חתימות. על כן, נשתמש ב-Virtual Time. במקום ביט שיציין 1/0, נשמור רפרנס ביט, ועוד מקום עבור חתימת זמן. חתימת זמן תעודכן בכל פעם שנעדכן את הרפרנס ביט. נניח שהזמן הנוכחי (הוירטואלי) הוא T, ואנו ניגשים לדף מסויים. בגישה אליו, נעדכן את החתימה להיות T, ונגדיל את הזמן הוירטואלי. במידה שאנו רוצים לבצע החלפה, אנו נבחר את הדף עם החתימה בעלת הזמן הנמוך ביותר. ככה אנו מקבלים את הדף שלא השתמשו הכי הרבה זמן, פר הרווח של הטיפול ב-Page Faults כלומר פר הרווח של החומרה.

ניזכר כי מלבד הרפרנס ביט שמרנו גם Dirty Bit אליו צריך להתייחס. דבר זה מוביל לאלגוריתם הבא של החלפת הדפים, שמהווה קירוב טוב ל-LRU.

```
1  void improved_clock_time(){
2          look at the page with the arrow
3          if (R==1) {
4                  advance the arrow, go to line 1
5          }
6          else if (R == 0 && (current virtual time) - (time of last use) < k){
7                  advance arrow, go to line 1
8          }
9          else if (dirty){
10                 advance arrow, go to line 1
11         }
12         else{
13                 evict page ( if not candidate is found, evict the oldest page (clean or
                       dirty) )
14         }
15 }
```

6.6.1 Global & Local Paging

עלינו להחליט האם כאשר תהליך גורם ל-Page Fault מערכת ההפעלה תמפה לו Page מאחד הדפים שלו (Local), או מאחד הדפים של תהליך אחר (Global)?

שימוש ב-Global Paging מאפשר לבחור את המועמד הטוב ביותר להחלפה. מעבר לכך, הוא מאפשר לנו להקצות לתהליכים דפים לפי תדירות השימוש בהם, כמו כן, זה חוסך את הליך הקביעה של "כמה דפים להקצות לכל תהליך", שכן זה לא קריטי. שימוש ב-Local Paging מגביל אותנו, אך מקל על בחירת הדף. לעומת זאת, הוא דורש בחירה מראש של כמה דפים להקצות, שזה יקר יותר.

6.7 ביצועים

אם p היא ההסתברות לקבלת Page Fault אז הזמן האפקטיבי לגישה לזכרון הוא ממוצע משקולל:

$$\text{Effective Access Time} = \text{p (Page Fault Time)} + (1-\text{p}) \cdot \text{(Memory Access Time)}$$

על כן, גורם ההאטה, הוא

$$\text{Slowdown} = \frac{\text{Effective Access Time}}{\text{Memory Access Time}}$$

אם $p = 0.001$ אז ה-$\text{SlowDown} = 250$, שזה מזעזע. זה נובע מכך שגישה לזכרון היא בסדר גודל של מאה נאנו שניות, ו-Page Fault לוקח כרבע מילי שנייה.

מה שביצענו כאן זה ניתוח Amortized, שכן הסתכלנו על ממוצע.

6.8 מקומיות זמנית (Temporal Locality)

מקומיות בזמן, אינטואיטיבית, היא גישה לאותו משאב ברצף, למשל

$$(\text{A, A, A, A}), (\text{B, B, B}), (\text{C, C, C, C, C})$$

אך בפועל, הגישה היא לפי משאבים פופולריים:

$$(\text{B, A, B, B, B, B, C, C, B, B, B, A, B, B, B, D, B, C, B})$$

בדרך כלל כמקבלים מידע, ורוצים לדעת איך הוא מתנהג, מניחים שמדובר בהתפלגות אחידה, אך זה למעשה הניחוש הכי גרוע שאפשר לעשות. מדוע? נסתכל על רשימת המוצרים הנמכרים באמזון. בראש הרשימה נמצאים ספרי הארי פוטר. האם זה אומר שמחר הם לא יהיו שם? להפך, מחר יקנו אותם יותר. תופעה זו מכונה "Rich Become Richer".

כהתפלגות שמתארת תופעה זו היא Zpif.

6.9 מדידת מקומיות

6.9.1 התפלגות ה-Zipf

במאה הקודמת מיפו את התפלגות המילים בשפה האנגלית. קיבלו שהמילה הנפוצה ביותר היא "The", וכי מספר ההופעות $c(i)$ של המילה ה-i, מקיים את הקשר

$$c(i) \propto \frac{1}{i}$$

עד כדי קירוב. מכאן נובע כי

$$\log c(i) \propto -\log i$$

ולכן בסקאלה לוגריתמית, מתקבל קו ישר. עבור המקומיות, אנו מסיקים כי גישות מסויימות יקרו לעיתים רחוקות מאוד.

6.9.2 Stack Distance

אחת הדרכים למדוד מקומיות של תכנית היא באמצעות מחסנית. כדי לעשות זאת, נסרוק את מרחק הכתובות.

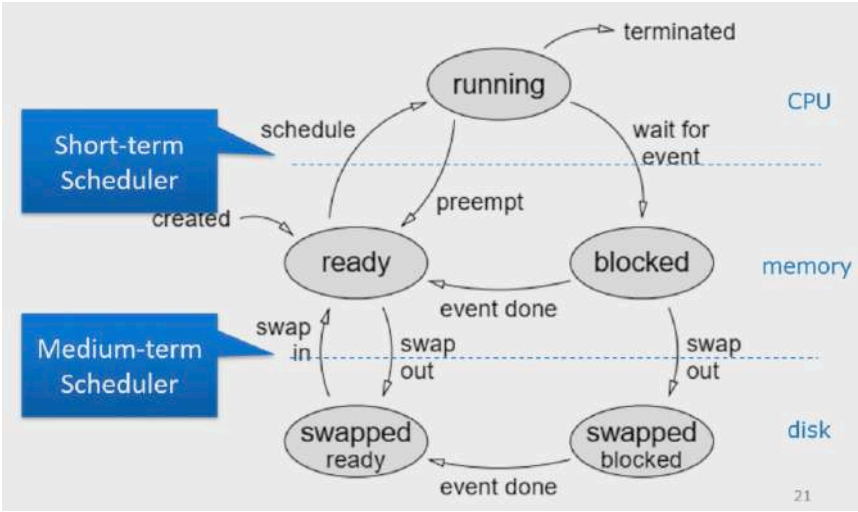
אם נמצא את הכתובות הנוכחית במחסנית, נשמור את העומק שלה, ונוציא אותה משם. לאחר מכן נדחוף אותה לראש המחסנית.

כך על פי העומק של הכתובת, נדע את רמת המקומיות שלה. למשל, אם לאחר גישה לכתובת 1, ראש המחסנית מכיל 1, הכתובת תהיה בעלת רמת מקומיות גבוהה, כי בכל הזמן שעבר ניגשנו רק אליה. אם היא בסוף המחסנית, תהיה לה רמת מקומיות נמוכה. ה-Stack Distance הוא המרחק מראש המחסנית.

את שיטה זו נוכל לשלב עם אלגוריתם ה-LRU, שמנסה לזרוק את מי שלא נגענו הכי הרבה זמן. נניח כי יש לנו זכרון של k מסגרות, איפה יהיה הדף שניגשנו אליו הכי פחות זמן, במחסנית? הוא יהיה במקום ה-k, בהכרח. לכן כדי לנתח את ההסתברות לקבלת Page Fault מספיק להסתכל על דפים בעומק גדול מ-k.

Streaming אחת הבעיות של מנגנון זה, היא שבמידה שנעבור על Stream של מידע שלא נכנס לכל הזכרון, אנו נעבור עליו באופן סדרתי, ונתקדם דף דף, אך היות שהמידע מאוד גדול, הדף שנבחר עלול להיות דף שניגש אליו מיד כשנסיים עם ערוץ המידע. כדי לטפל בזה, מוסיפים לכל דף סמן של "פעיל" או "לא פעיל" וכך ניתן לזהות האם כדאי לקחת את הדף. Linux משתמשת במנגנון זה.

MultiProgramming על אף שתכנות מקבילי מגביר משמעותית את ה-CPU Usage, כמו למשל, זכרון DMA. אך יותר מדי רכיבים הרצים במקבילים, דווקא יקטינו אותו מאוד. מדוע? רכיבים מקבילים דורשים זכרון, וכאשר כמות הזכרון עולה על כמות הזכרון שיש למחשב, יש Page Faults והרבה. לכן, כאשר תהליך ירוץ, חלק נכבד מהזמן יוכבד בכלל לטיפול ב-Page Faults ובפועל, זו תהיה הפעולה העיקרית של המעבד, מה שיוריד את ה-CPU Usage. תופעה זו נקראית Thrashing.
הפתרון הוא למנוע מחלק מהתהליכים להשתמש בזכרון. כלומר, בהליך התזמון נוסיף עוד שלב, והוא בחירה האם תהליך יכול לקבל זכרון, או שלא. המתזמן האחראי על ניהול זה נקרא ה-$\text{Medium} - \text{Term Scheduler}$:



איור 48: דיאגרמת התזמון החדשה. כאשר תהליך הוא במצב Swapped Blocked הוא רק בדיסק, כאשר הוא במצב Ready הוא מחכה יתנו לו זכרון. **נבחין כי מצב** Swapped Blocked **יכול לקרות רק לאחר מצב** Blocked**, כלומר לאחר ריצה. נבהיר שכל הדפים של התהליך נמצאים במצב זה בדיסק, ואין לו דפים ב-**Ram.

6.9.3 גודל טבלת הדפים

במערכת 32Bit, אם גודל הדף הוא 4KB, אז ה-offset בתוך הדף הוא 12 ביטים, ולכן כל תהליך בעל

$$2^{20} = \left(2^{10}\right)^2 \approx \left(10^3\right)^2 = 10^6$$

דפים. אז אם כל כניסה בטבלה היא 4B, דרושים $2^{20} \cdot 4$ בתים לטבלה, שזה 1000 דפים. לכן, רק בשביל הטבלה של תהליך אחד דרושים 1000 דפים בזכרון.

אבל, מה קורה כאשר זו מערכת של 64Bit?

אם גודל ההיסט הוא עדיין 12 ביטים, וכל כניסה היא 2^3Bit אז לכל תהליך 2^{52} דפים, ולכן הטבלה בגודל 2^{55} דפים. היות שכל דף בגודל 2^{12} מספר הדפים של הטבלה הוא 2^{43}. בפועל, מרחב הכתובות הלוגיות הוא 2^{48} בתים.

אם גודל הזכרון הפיסי הוא $32\text{GB} = 2^{35}$ אז יש 2^{23} דפים ולכן בפועל $2^{43} - 2^{23}$ מהדפים בטבלת הדפים, בכלל לא חוקיים, שזה פי מיליון בערך ממספר הדפים החוקיים.

לכן צריך לחשוב כדי לממש טוב יותר את טבלת הדפים, מבלי לבזבז זכרון.

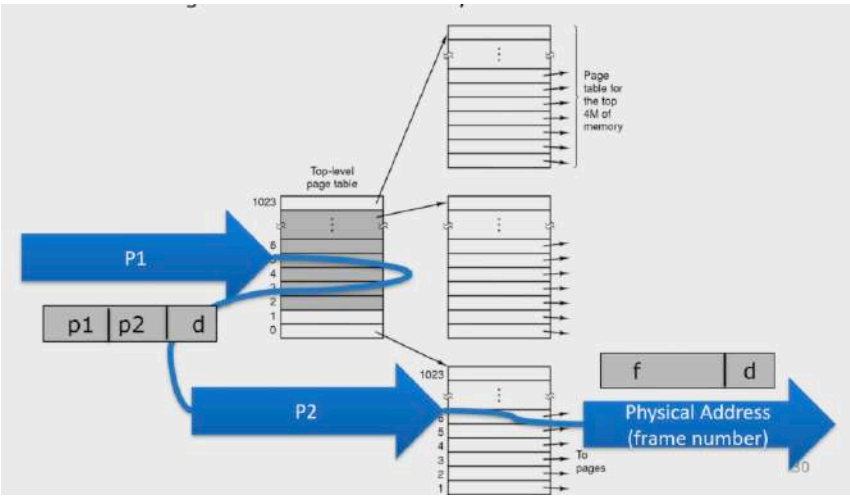
6.10 סוגי טבלות דפים

6.10.1 טבלות דפים היררכיות

הפתרון לבעיה שהוצגה קודם הוא שימוש בטבלות דפים היררכיות, כמו שראינו בעבר. אנו נחזיק טבלה של טבלות, כך נמשיך עד עומק מסויים. תא בטבלה יכיל 0 אם התא לא מחזיק עוד טבלה, וכך לא נצטרך לאחסן טבלות של דפים לא קיימים.

הגישה תתבצע בשני שלבים. עבור מידע (p_1, p_2, d) אנו ניגש לטבלה הראשית במיקום ה-p_1, לאחר מכן לטבלה שהתקבלה, אבל במיקום ה-p_2 ולבסוף למיקום ה-d של הדף שהתקבל. התא בטבלה יכיל את המסגרת שמכילה את הטבלה, וכך למעשה, הטבלות מכילות את כל המסגרות שמכילים אותן. גישות לדפים לא קיימים לא יתבצעו ולכן לא נצטרך לשמור טבלות עבורם.

במקרה שלנו, במקום טבלה 1 עם 1M כניסות, נחזיק $\underbrace{1}_{\text{טבלא שמכילה את המסגרות של הטבלאות}} + \underbrace{1024}_{\text{טבלאות דפים}}$ טבלות, כל אחת עם 1000K כניסות, כאשר חלק מהן בכלל לא יוחזקו בזכרון.



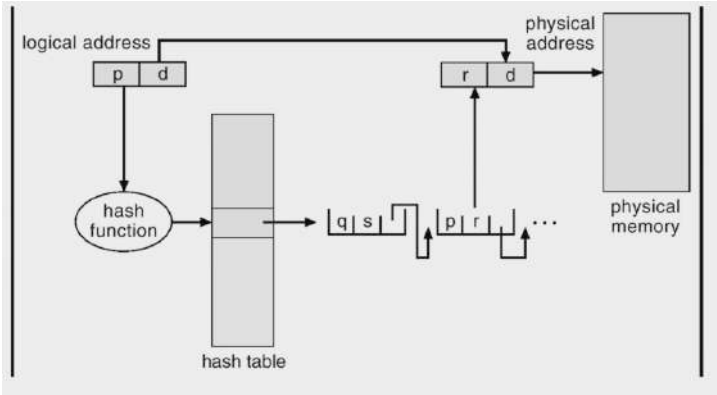
איור 49 : המחשה לטבלת הדפים ההיררכית.

יש לנו כאן טרייד אוף של אחסון טבלות אל מול טבלות שלא צריך.

6.10.2 Hashed Page Tables

המטרה שלנו היא לכווץ את מרחב המפתחות הוירטואלי למרחב מפתחות קטן יותר ולכן דרך הגיונית היא להשתמש בפונקציית גיבוב H.

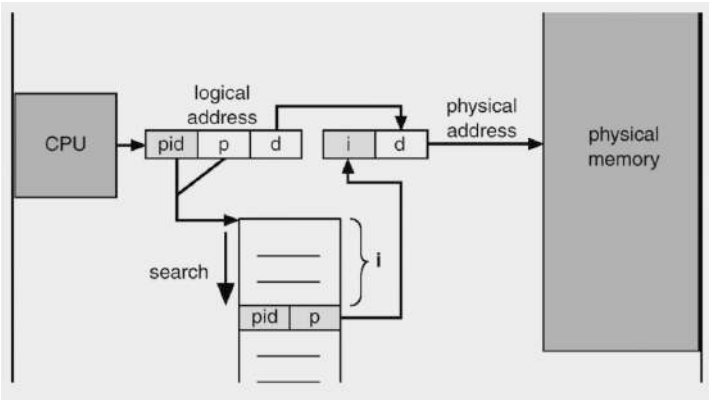
בהנתן זוג (p, d) נסתכל על הטבלה במיקום ה-$\mathrm{H}(p)$ ונעבור על הרשימה המקושרת שנמצאת שם. כל איבר ברשימה יכיל זוג (q, s), כאשר $q = p$ נדע שהגענו לתא המתאים לנו, ולכן הכתובת תהיה (s, d). הסיבה שיש שם רשימה, היא שיתכן כי כתובות שונות מופו לאותו תא בטבלה. זה המקרה בו האיבר מופה לטבלה, אם הוא לא מופה, אז יש Page Fault :



איור 50 : המחשה לטבלת הדפים עם פונקציית הגיבוב.

6.10.3 Inverted Page Tabels

בטבלת דפים הפוכה, במקום לשקף את הזכרון הוירטואלי, אנו משקפים את הזכרון הפיסי, וכך נכלול אך ורק את הדפים הקיימים. בכל תא, לא נוכל להחזיק כתובת של זכרון וירטואלי בלבד, אלא גם את התהליך שהיא שייכת אליו, שכן תהליכים שונים ברגע נתון עלולים לרצות את אותו הזכרון הפיסי, אך רק לאחר הוא שייך. לכן בכל תא בטבלה יהיה זוג $(\mathrm{pid}, \mathrm{p})$, ועבור שלשה (pid, p, d) נחפש בטבלה באופן סדרתי את הזוג (pid, p) ואם מצאנו, נדע שהמסגרת היא האינדקס בטבלה, כלומר עבור התא ה-i נקבל שהכתובת הפיסית היא (i, d). זה אמנם לינארי, ויקר, אך חסכוני בזכרון.



איור 51 : המחשה לטבלת הדפים הפוכה.

6.10.4 השוואה

מאפיין	טבלת דפים היררכית	טבלת דפים מגובבת	טבלת דפים הפוכה
משקפת את הזכרון	הוירטואלי	בין לבין, יותר מהפיסי, אך הרבה פחות מהוירטואלי	פיסי

6.11 הגנה ושיתוף

לניהול הזכרון שראינו עד עכשיו יש יישום מאוד חזק, והוא הגנה על הזכרון. היות שטבלת הדפים מכילה את כל הדפים של התהליך, כל דף שלא נמצא בטבלה, לא קיים מבחינתו, ולכן הטבלה מפרידה את התהליכים אחד מהשני. מעבר לכך, ניתן ליצור סגמנטציה.

כל סגמנט יכיל טבלת דפים משלו, שיהיו בעלי משייך לסגמנט, וכך גישה מתוך הסגמנט לדף שאיננו בטבלה, תוביל לשגיאת סגמנטציה. כך למשל נוכל לבצע הפרדה של ה-

$$\text{Heap, Stack, Code Seg, Data Seg}$$

ולקבל הגנה מפני גישות לא חוקיות לסגמנטים, מתוך סגמנט.

מלבד זאת, נוכל לאפשר שיתוף של זכרון בין תהליכים. בעוד דפים שלא מופיעים בטבלה לא קיימים, דפים שמופיעים בשתי טבלות של תהליכים שונים, קיימים עבור שני התהליכים, ולכן מהווים זכרון משותף, לכן על ידי סנכרון, מתאפשר להם לשתף נתונים.

6.12 סיכום

הדיון שלנו על ניהול זכרון מסתכם בטבלה הבאה :

בעיה	פתרון
נקודת מבטו של תהליך שונה מהמציאות	תרגום כתובות : base + bound, Page Tables
פרגמנטציה חיצונית	דפדוף
מרחב הכתובות הוירטואלי גדול ממרחב הכתובות הפיסי	Virtual Memory ו-Swapping

7 מערכות קבצים

7.1 מבוא

כשדיברנו על הפשטות שמערכת ההפעלה מבצעת, הזכרנו קבצים - במקום שניגש בדיוק למיקום בדיסק שבו המידע נמצא, ה-OS מאפשרת לנו לגשת אליו באופן מאוד נוח.

עולה השאלה, מהו קובץ מעבר להפשטה?

אנו יודעים שהוא רצף מידע חסום, ושיש לו הרשאות גישה, אך מה מבדיל אותו מזכרון של תהליך?

זכרון הקובץ, נמצא בדיסק, ולכן, הוא לא נמחק כשמהחשב נכבה.

נהוג להדביק למונח את הרצף הבא:

$$\text{File} = \text{named presistent sequential data storage}$$

- סדרתיות: המידע בקובץ הוא מסודר.
- השארות לאורך זמן: נוכל לכבות את המחשב למשך שנים, וכשנפתח אותו, המידע עדיין יהיה שם.
- מתן שמות: ניתן למצוא אותו באמצעות שם.

מתן שמות השמות הם חיצוניים ביחס למערכת ההפעלה, וקיימים בהקשרים רבים. למשל, אם אנו יוצרים סגמנט חדש עבור מידע משותף של תהליכים, צריך לתת לו שם, כדי שהתהליכים ידעו לאיזה סגמנט לגשת, או, אם אנו רוצים לגשת לפורטים, צריך שיהיה להם שם. כמו כן, עלינו לזכור את שם הקובץ, או למצוא אותו ברשימה, שעלולה להיות ארוכה מדי. כדי להימנע מחיפוש ברשימה, נהוג לחפש לפי תוכן.

השארות לאורך זמן קבצים נשמרים ברכיבים שאינם נדיפים (non volatile), ולכן שורדים את מוות התהליך שיצר אותם. הם נשמרים על ה-SSD/Disk, בשונה מזכרון של תהליך שמת עם סיום ריצתו. השארות לאורך זמן הוא מושג יחסי. בפועל, לא מדובר בהשארות לנצח. למשל, ניירות פפירוס ממצרים העתיקה נשמרים כבר 2000 שנה ואפשר לקרוא אותם, אבל מידע במדיה החברתית, עלול להתפוגג עוד $10 - 20$ שנה, כיוון שכל ביט במידע, נשמר על מספר לא גדול של אטומים, ותזוזה שלהם, עלולה לגרום להתפוגגות. אפשרות אחרת, היא שפורמט המידע לא ידוע, למשל, שימוש בתמונת BMP, שמעתם עליה? היא אמנם לא ישנה במיוחד, אבל לא בשימוש כמו jpeg/png, ולכן אם תקבלו אחת, לא תדעו איך לעבד אותה, על אף שהמידע נמצא בידיכם, ואתם יודעים מה הוא אמור לייצג.

מבנה הקובץ ב-Unix קובץ הוא רצף של בתים שיכול לייצג טקסט, תמונה, וידאו, צליל, תכנית הרצה, קוד בינארי, מידע נומרי, ואף קובץ שמסתיר קובץ אחר. אך אפשר גם שקובץ יהיה אוסף של רשומות כמו ב-IBM, ואז כשניגשים לקובץ מבקשים את הרשומה ה-i.
אפשרות נוספת היא שהוא יהיה מורכב מזוגות של $\text{attribute} - \text{value}$ כמו ב-Windows NTFS, שם כל זוג ייצג מידע רלוונטי בקובץ, למשל שדה של אייקון, שהערך שלו יהיה הפיקסלים שיוצגו על המסך כדי להציג את הקובץ, או $\text{attribute} = \text{zone}$ שהערך שלו הוא האזור ממנו הגיע הקובץ, למשל האינטרנט, בנוסף הוא יכיל שדה של נתוני הקובץ, מערכת ההפעלה יודעת לטפל בשאילתות לשדות ולמידע.

מה לא לעשות - Extension אנו מפרידים בין שם הקובץ לבין התוספת. התוספת היא סוג הקובץ, למשל "cpp". ב-Unix לסיומות אין משמעות, ומערכת ההפעלה לא מכריחה לשים אותם, אך יישומים עלולים לדחות קבצים בעלי סיומות לא מתאימות, על אף שהדבר לא באמת מייצג, שכן סוג הקובץ נקבע על ידי המופיעים בתחילת הקובץ. לכן ניתן לקרוא ל-"file.c" כ-"file.jpg" וזה לא שימוש טוב.

ההבדל בין היבט מערכת ההפעלה להיבט האפליקציה, הוא שהאפליקציה מציגה לנו את תוכן הקובץ "היבש", כלומר את הבתים שנמצאים בו, כמו כן היא אחראית על אחסון המידע, שמירה על מידע רלוונטי עליו, **ומונעת הגבלה של האפליקציה**. האפליקציה אחראית לתרגום הבתים - אם זה לטקסט, או ל-hex והחלטה מה משמעות המידע. כלומר מערכת ההפעלה היא מתווכת של האפליקציה במקרה זה.

עלינו להבין את:

- ממשק מערכת הקבצים (System Calls)- איך מבקשים קריאה, כתיבה והרצה, יצירה, מתן שמות, שיתוף והגנה.
- מימוש מערכת הקבצים - איך שומרים את המידע על הדיסק, איך מקצים זכרון ומנהלים אותו, ואיך שומרים על ביצועים טובים.

7.1.1 פעולות מערכת הקבצים

נראה פעולות בסיסיות על קבצים:

- יצירה.

- מחיקה.
- פתיחה.
- סגירה.
- כתיבה.
- קריאה.
- הרצה.
- seek - תזוזה בתוך הקובץ.
- חתיכת קובץ (Truncate).
- set - שינוי השדות של הקובץ.
- get - קבלת השדות של הקובץ.
- שינוי שם של קובץ.

תמיכה במערכות קבצים נבהיר כי מערכת ההפעלה צריכה לתמוך בכמה סוגים שונים של מערכות קבצים. כלומר עליה להיות מסוגלת לקבל מערכת קבצים חיצונית ולתקשר איתה. למשל, שימוש ב-USB, אשר בו יש מערכת קבצים שנכתבה על ידי היצרן, שאינה בהכרח תואמת את המערכת הקבצים שיש על המחשב שלנו. למרות זאת, אנו מסוגלים לחבר אותו למחשב, ומערכת ההפעלה יכולה לעבוד עם מערכת הקבצים שלו. כמו כן, DVD מכיל גם מערכת קבצים משלו, והיא שונה ממערכת הקבצים של ה-USB, שכן שם צריך לתזמן את ההגעה של הדיסק. בכל מקרה, מדובר בזכרון לא דליף, שכן הוא נשמר לאחר ניתוק מהמחשב. Linux למשל תומכת ביותר מ-40 מערכות קבצים שונות.

7.1.2 שדות של קובץ

לקובץ שדות שונים, וחלקם ניתנים על ידי מערכת ההפעלה (METADATA), ותלויים בסוגה.

המשתמש מה שמטריד את המשתמש הוא שם הקובץ, גודל הקובץ, הרשאות הגישה שלו, ותאריכים שנשמרו בקובץ.

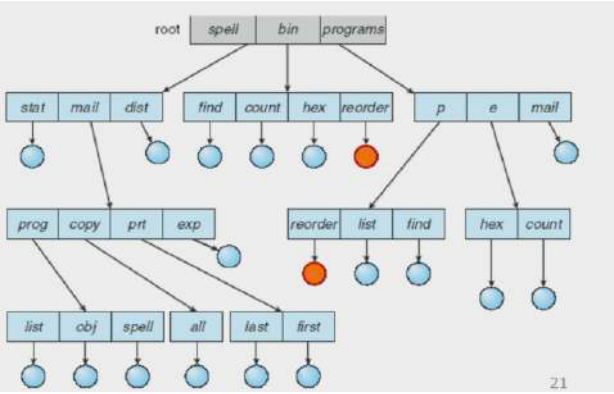
מערכת הפעלה מערכת ההפעלה נותנת לכל קובץ תווית מיוחדת כדי לזהות אותו בתוך מערכת הקבצים. כמו כן, היא משמרת את המיקום הנוכחי שלנו בתוך הקובץ, כלומר, פוינטר למיקום בקובץ שהגענו אליו.

7.2 מרחב השמות של קבצים (Namespace)

הקבצים מאורגנים באופן היררכי באמצעות תיקיות שמטרתן לשמר:

- יעילות - קל למצוא את מיקום הקובץ, ונמנעים מחיפוש בתוך רשימה ארוכה.
- הפרדה - נוח למשתמש, שני משתמשים יכולים ליצור שני קבצים בעלי אותו שם. בנוסף, כל קובץ יכול להכיל כמה שמות שונים.
- קיבוץ - סידור לוגי של הקבצים על פי תכונות מסויימות. למשל, כל הקבצים עבור משחק מסויים יהיו תחת אותה תיקייה, או כל הקבצים של OS/Ex4.

בשיטה זו, מה שמבדיל קבצים אחד מהשני זה המסלול מהשורש אל הקובץ. למשל, שני קבצים באותה תיקיה לא יוכלו להיות בעלי אותו שם, שכן **שמות** הם **ביחס** לתיקייה.



איור 52: עץ התיקיות במחשב. ניתן לראות שיש שני קבצים שונים עם אותו השם, שכן מה שמבדיל אותם זה המסלול מהשורש.

7.2.1 תיקיות (Directories)

תיקיות ממפות קבצים לאובייקטים של קבצים, כלומר, הם רמה אחת בהיררכיה של מתן השמות.

המידע למיפוי נשמר כקובץ, ולכן תיקייה היא למעשה קובץ (!!!). מי ששולט על תוכן התיקייה היא מערכת ההפעלה, שכן האפליקציה היחידה שמשתמשת בה היא מערכת ההפעלה עצמה, ולכן היא זו ששולטת עליה.

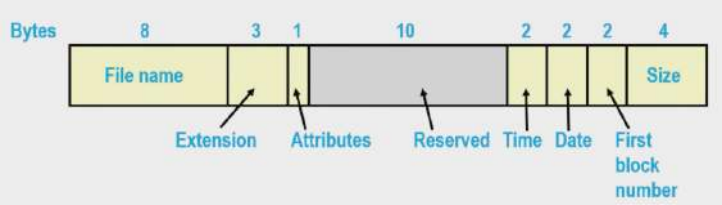
על כן, מערכת ההפעלה משתמשת במבנה נתונים יעיל שיאפשר לחפש בתוך התיקייה (שהיא קובץ) את הקבצים שהיא ממפה, ליצור קבצים חדשים, למחוק אותם, להחזיר רשימה של כל הקבצים בתוכה, ולשנות את השם שלהם.

מערכת הקבצים צריכה להחליט איך היא מארגנת את שמות הקבצים בתוך קובץ התיקייה. נראה כמה דרכים לבצע זאת:

אוסף של רשומות נשמור את המידע כאוסף של רשומות, כאשר כל רשומה תכיל את השם של הקובץ, ואת כל המידע שאנחנו צריכים לדעת עליו (שדות הקובץ). כל כניסה בטבלה שנשמור היא מגודל קבוע, הכתובות של הקובץ בדיסק ישמרו בטבלה כדי שנוכל לגשת לקובץ. מערכת ההפעלה הקדומה $MS-DOS$ השתמשה בשיטה זו.

מיפוי שמות לאובייקטים נשמור טבלה שתכיל את שמות הקבצים, רק שליד כל שם, לא יהיו שדות הקובץ, אלא מצביע על אובייקט של הקובץ, שגישה אליו תאפשר לנו לבצע את הפעולות שנרצה. האובייקט הוא מבנה נתונים שלא נשמר בתיקייה אלא כחלק מהקובץ, כך שמה שנמצא בתיקייה זה רק המיפוי מהשם למצביע.

דוגמה. $(MS-DOS)$ באיור הבא ניתן לראות כיצד שומרים את המידע בתיקייה כאוסף של רשומות:



איור 53: ניתן לראות שנשמר שם הקובץ, וכי נשמר פוינטר לבלוק הראשון שמכיל את המידע של הקובץ בדיסק. במילים אחרות, נשמרים כל השדות הרלוונטים, בתוך התיקייה.

דוגמה. $(UNIX\ V7)$ באיור הבא ניתן לראות כיצד שומרים את המידע על ידי מיפוי לאובייקט:

איור 54: מלבד השם, נשמר אך ורק פוינטר לאובייקט של הקובץ, אשר נקרא $I\,(index)-node$. כל שאר השדות נשמרים באובייקט, ולכן כדי לקבל אותם פשוט ניגש לאובייקט.

ב-Linux הוסיפו עוד שדה של type - שמציין האם מדובר בתיקייה או קובץ. מדוע? שימוש בפקודה ls מציג לנו את עץ התיקיות החל מהתיקייה הנוכחית, וכדי לדעת האם מדובר בתיקייה או קובץ, אנו נדרשים לקרוא את המידע מה-$I-node$. הבעיה היא שהמידע נשמר בדיסק, וזה מאוד איטי $(Mili-Seconds)$ שזה כשבועיים עבור המחשב.

נתיבים לכל מערכת קבצים יש תיקיית שורש, אך לכל תהליך יש תיקייה בה הם עובדים, ועבורם, זו תיקיית השורש. כלומר הגישה לקבצים בתהליך היא רלטיבית לתיקיית העבודה שלהם. לכן, על מנת לגשת לקובץ, התהליך חייב לספק את הנתיב עבור הקובץ, שיכול להיות:

אבסולוטי: נתיב מהשורש לתיקייה.

רלטיבי: נתיב מתיקיית הבית של **התהליך**.

כמו כן, רכיבי הנתיב מופרדים על ידי / ב-UNIX וב-\ ב-Windows. בנוסף, ברוב מערכות ההפעלה יש מציינים מיוחדים:

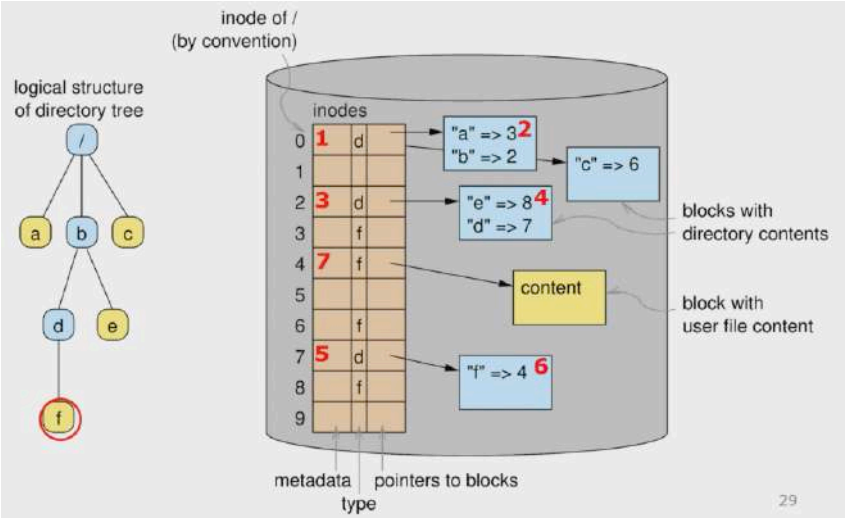
"." מציין את התיקייה הנוכחית.

".." מציין את תיקיית האב.

כאשר מתקבל נתיב לקובץ, יש צורך לפרש אותו (parse), ובכל פירוש של גורם בנתיב, נדרשות שתי קריאות מהזכרון.

1. קריאת המיפוי בתוך התיקייה בו הוא נמצא.
2. קריאת המידע בתוך האובייקט שלו.

באיור הבא נראה כיצד הפירוש מתבצע.



איור 55: הנתיב מתחיל מתיקיית השורש "/" ומשם מתחיל החיפוש. המוסכמה היא שה-$\mathrm{Node}-\mathrm{I}$ של תיקייה זו הוא מספר 0, על מנת שיוכלו לדעת בכל עת איפה היא נמצאת. נעקוב אחר הנתיב הכחול. תחילה אנו מחפשים את b בתוך התיקייה, ורואים שהוא מתאים ל-$\mathrm{Node}-\mathrm{I}$ מספר 2. אנחנו עוברים ל-$\mathrm{Node}-\mathrm{I}$ מספר 2 ומחפשים שם את d ורואים שהוא ב-$\mathrm{Node}-\mathrm{I}$ מספר 7, לכן עוברים לשם, ומחפשים את הקובץ f שנמצא ב-$\mathrm{Node}-\mathrm{I}$ מספר 4 ניגשים לשם ומקבלים את התוכן של הקובץ.
ניתן לראות שבכל מעבר על רכיב, אנו אכן מבצעים שתי קריאות מהזכרון - אחת עבור התיקייה הנוכחית, ואחת עבור האובייקט.

7.2.2 הרשאות גישה

כל תהליך זה כפוף להנחה שמותר לנו לבצע אותו. כלומר, אנו מגנים על כל קובץ על ידי הוספת הרשאות גישה מסוג

$$\mathrm{Read/Write/Execute/Append/Delete/List}$$

המימוש של מתן הרשאות גישה משתנה בין מערכות הפעלה. למשל, נרצה להיות מסוגלים לחסום קבוצות של משתמשים, באופן יעיל, ולא על ידי חסימת "משתמש משתמש". לדוגמא, במערכת הקבצים של האוניברסיטה, נרצה לחסום את כל הסטודנטים. לכך יש שתי גישות:

Minimal כמה ביטים בודדים לסימון הרשאות, המאפשרים הרשאה למשתמש יחיד, קבוצה יחידה וכל השאר. דבר זה נוח כי הוא מכיל כמה בתים בודדים. הבעיה היא שהוא לא מאוד אקספרסיבי. משתמשים בשיטה זו ב-UNIX/LINUX. אופן מתן ההרשאות הוא על ידי מטריצה 3×3 של $0, 1$, כאשר כל שורה מייצגת הרשאות לרכיב אחר: בעלים, קבוצה, וכל השאר. למשל:

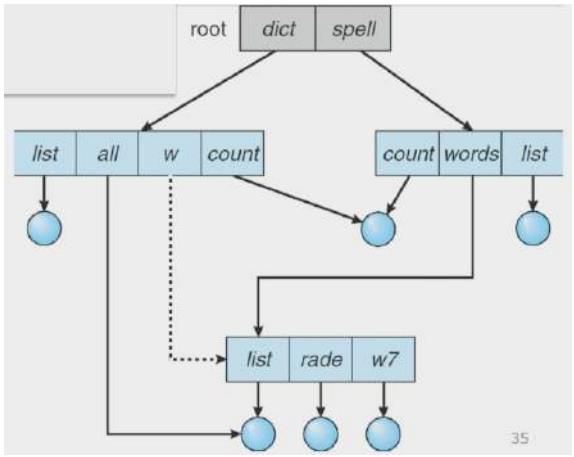
	R	W	X	(Octal Base)
Owner Access	1	1	1	7
Group Access	1	1	0	6
Others Access	0	0	1	1

או אם נרצה לשנות הרשאות הקובץ להרשאות המוצגות בדוגמא נשתמש בפקודה chmod 761, הספרה הראשונה מתייחסת לבעלים, השנייה לקבוצה, והשלישית לכל השאר. ניתן לשנות את הבעלים, אך הדבר מסוכן, שכן לא נוכל לקבל בעלות על הקובץ אלא אם הבעלים החדש יחליט אחרת, ניתן גם לשנות את הקבוצה.

Detailed לקובץ רשימה שמתארת בדיוק מי יכול לבצע כל דבר עלה קובץ. זה מאוד אקספרסיבי, אבל קשה לייצוג באופן קומפקטי. מערכת ההפעלה שמשתמשת בשיטה זו היא Windows.
הדרך בה ממשים זאת היא אכן באמצעות רשימה ((Access Control List (ACL), שבה כל כניסה דומה למטריצה במקרה הקודם, שמכילה זיהוי לקבוצה ולמשתמש ופעולה שעליה כתוב האם מותר להם לבצע או לא.
קיימים כללים מוגדרים לאיך לפענח האם פעולה מסויימת אפשרית או לא עבור המשתמש או הקבוצה. כדי לעשות זאת, עבור אובייקט במערכת הקבצים, נבדוק אם יש לו ACL. אם אין לו, זה אומר שאפשר לבצע עליו הכל. אחרת, אם הוא Null, כלומר ריק, אי אפשר לבצע עליו דבר. אחרת, אם יש לו (Access Control Entries (ACEs, נשתמש בכניסה הראשונה ששייכת למשתמש או לקבוצה שהוא משתייך אליה,
כאשר כניסות האוסרות על משתמשים לבצע פעולות נמצאות ראשונות בתור, ואחריהן כניסות המאפשרות.
במידה שלא נמצאה כניסה רלוונטית, כלומר הגענו לסוף הרשימה, לא תנתן הרשאה לביצוע הפעולה. הסיבה לכך שיש עדיין כניסות עם איסורים על פעולות, היא כדי לחסוך במעבר על הרשימה. הרי אם אפשרנו לקבוצה לבצע פעולה מלבד אחד, אשר עליו אסרנו לבצע אותה, הוא יעבור על האיסורים בתחילת הרשימה, ולא יצטרך לעבור על כל האישורים בהמשך.

עתה, נבין כיצד קבצים יכולים להכיל כמה שמות שונים:

- Hard Links - במקרה זה יש לנו שתי תיקיות שונות, שלהן קובץ עם אותו שם שמצביע לאותו מבנה נתונים של הקובץ.
- Symbolic Links (Soft Links) (ln − s) - יש לנו קובץ בתיקייה, וקובץ בתיקייה אחרת שהתוכן שלו הוא הנתיב לקובץ המקורי. כלומר, הוא לא מצביע לאותו אובייקט, אלא מקשר לקובץ המקורי, כאשר פותחים אותו, מגלים שהוא Symbolic Link ומגיעים לקובץ המקורי.



איור 56: Count נמצא בשתי תיקיות שונות כ-Hard Link.
w, list הם דוגמא ל-Soft Link.
אחת הבעיות ב-HardLink היא ששינוי הרשאות גישה בצד אחד, לא משפיע על הרשאות הגישה בצד האחר. ב-Soft Link זה לא ככה כמובן.

7.3 Layout and Access

בשלב זה נרצה להבין כיצד נשמרים הנתונים של הקבצים על הדיסק וכיצד מתבצעות הגישות.

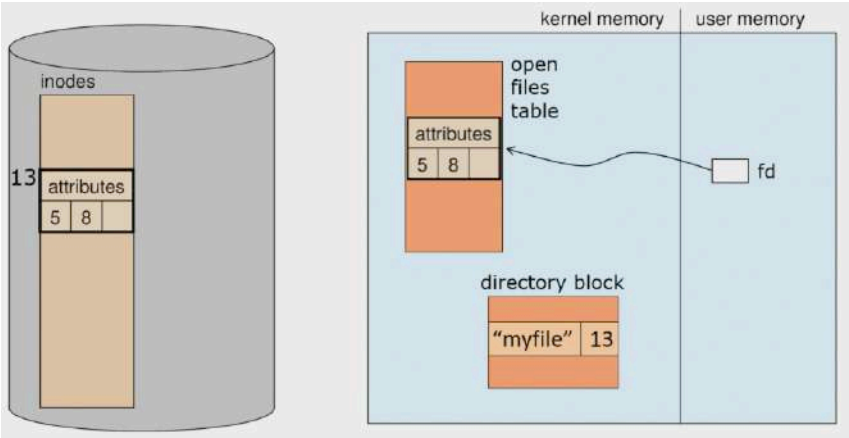
שאלה למה צריך לפתוח קובץ? הרי אפשר לבדוק אם הוא קיים בביצוע הפעולה שנרצה.

נבחין כי ביצוע הפעולה דורשת פירוש של הנתיב, מה שדורש בכל פענוח לפחות שתי גישות לדיסק, שזה דבר יקר, לכן באמצעות פתיחה של הקובץ, הפירוש מתבצע פעם אחת, ואז אפשר לשמור את המידע באובייקט המתקבל במקום ייעודי ב-Cache המכונה "Open Files Tables".

שאלה למה לסגור קובץ? הרי יתכן שנרצה להשתמש בו שוב בעתיד, וחבל לסגור אותו, מה גם שמערכת ההפעלה תדאג בסיום ריצת התהליך לסגור את הקבצים שלא נסגרו.

יחד עם זאת, טבלת הקבצים הפתוחים ב-PCB היא בגודל חסום, ולכן אם לא נסגור את הקבצים, היא תתמלא ולא נוכל ליצור קבצים חדשים.

```
fd = open("myfile"); // "myfile" is inode 13
```



איור 57: המידע שנשמר ב-fd הוא פוינטר לטבלת הקבצים בתהליך שמכילה את שדות הקובץ, ביניהם - פוינטר למיקום הנוכחי בקובץ, מספרים שהמייצגים באילו בלוקים נתוני הקובץ נמצאים.

7.3.1 שמירת קבצים בדיסק

אם נשמור את הקבצים בדיסק בצורה רציפה, אנו עלולים לקבל בעיות פרגמנטציה, וגם אם נטפל בהן, נתקבל בבעיה נוספת. נניח שהקבצים נמצאים בסדר הבא:

Empty
A
B
C

אם נרצה להגדיל את C לא נוכל, שכן בהמשך הזכרון נמצאים כבר A, B.

לכן, נוכל להשתמש בשיטת ה-Compaction - נעתיק קבצים מחדש בדיסק כדי לסדר את המידע באופן מיטבי. זו פעולה יקרה ומאוד בעייתית, שכן לא תמיד יש מקום כדי להעתיק את המידע.

דרך נוספת היא שימוש ב-Paging - לחלק את הזכרון למסגרות בגודל קבוע (4KB) למסגרות אנו קוראים Blocks. נקודת המבט של מערכת הקבצים היא שכל קובץ מחולק לבלוקים בגודל קבוע, זו נקודת מבט וירטואלית. בפועל, הדיסק מסתכל על הקובץ כאוסף של סקטורים, כאשר כל סקטור הוא בגודל 512Bytes.

כאשר אנו מבקשים ממערכת ההפעלה לתת לנו מספר בתים מקובץ, היא לעולם לא תתן רק את בתים אלה, אלא תיקח מהדיסק בלוק שלם לפחות. מתוך הבלוק היא תתן את המידע. או, אם המידע מתפרש על כמה בלוקים, היא תעתיק את שניהם, לכן עליה להיות מסוגלת למפות כל בקשה לבלוקים, והיסט בתוך הבלוקים.

למשל, getc, putc קוראות ומדפיסות בית אחד, אך מערכת ההפעלה תקרא עבורן את כל הבלוק, וממנו תעתיק את הבית הרלוונטי, לכן לא מומלץ לבצע קריאה רציפה של בתים עם getc, putc, שכן מתבצעת ההעתקה כמה פעמים, וזה מיותר.

הערה. כדי לבצע דבר זה צריך לקבוע מבנה נתונים רלוונטי לארגון המידע, להתחשב בהתפלגות של בלוקים שונים ועוד.

דוגמה. (קריאה) נניח שאנו משתמשים בממשק המשתמש של הקבצים, ומבקשים לקרוא N בתים מהמיקום הנוכחי בקובץ:

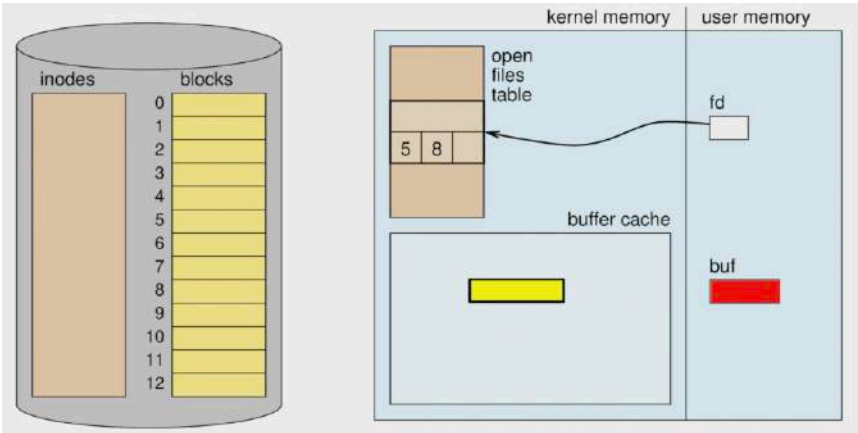
read (fd, buf, N);

מה יקרה בפועל?

מערכת ההפעלה תזהה אילו בלוקים צריך להעתיק כדי לקרוא את הבתים.

היא תקרא את הבלוקים.

תעתיק את המידע הרלוונטי ל-buffer.



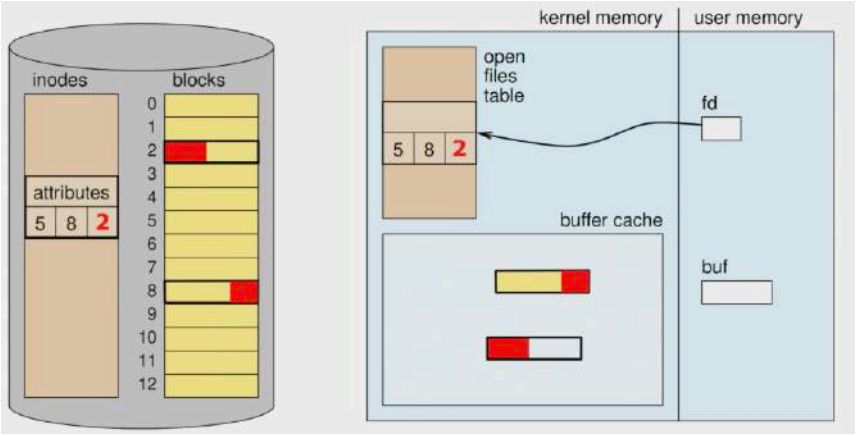
איור 58: תחילה אנו קוראים את הבלוקים הרלוונטים ומעתיקים אותם ל-buffer cache. אנו לא מעתיקים את כל הבלוק ל-buf, שכן הוא ידרוס את מה שנמצא אחריו בזכרון, אלא רק את המידע הרלוונטי המסומן באדום.

דוגמה. (קריאה) נניח שאנו מבצעים את הפעולה

write (fd, buf, 3000) ;

תחילה נזכור כי מערכת ההפעלה קוראת בלוקים, ולא חלקי בלוקים, ולכן, לא נוכל פשוט לכתוב מספר בתים אל בלוק קיים, אלא נצטרך להעתיק את הבלוק ל-cache, להעתיק אליו את המידע, ואז להעתיק אותו חזרה לדיסק. פעולה אחרת, כמו, יצירת בלוק ב-cache העתקת המידע אליו, והעתקה

לדיסק, תדרוס את המידע הקודם בבלוק, לכן לא נעשה זאת. עתה, יתכן שהבלוק כמעט מלא, ואנו צריכים לעבור לבלוק הבא, אבל הפלא ופלא, נגמרו הבלוקים המוקצים לקובץ. לכן, אנו נצטרך ליצור בלוק חדש. כדי לעשות זאת, אנו לא צריכים להעתיק בלוק מהדיסק, שכן מדובר בבלוק חדש, ולכן במקום זאת, אפשר להקצות מקום ב-cache, להעתיק אליו את המידע, ואז להעתיק אותו לדיסק. כמו כן, עלינו לעדכן את טבלת הקובץ על כך שהוספנו עוד בלוק, אבל גם על השינוי בגודל הקובץ, לכן לאחר שנעתיק את הבלוקים, נעתיק את השדות החדשים ל-inode.



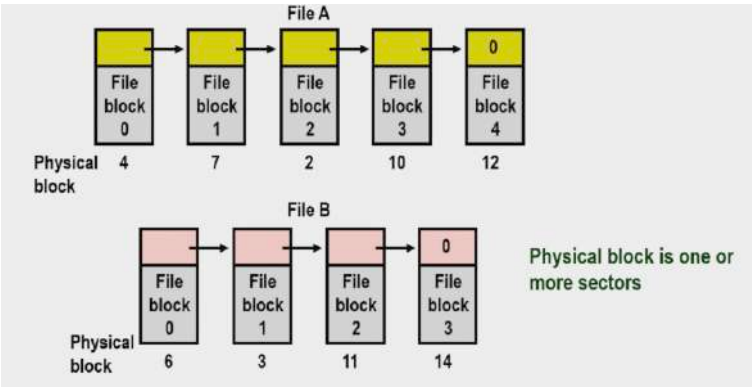
איור 59 : תחילה אנו מעתיקים את הבלוק הצהוב, ומעתיקים אליו בתים שהם המידע האדום. לאר מכן אנו מעתיקים את התוצאה לדיסק.
עבור המידע שנשאר, דרוש עוד בלוק, ואותו הקצאנו (בלוק כחול ריק) ב-buffer cache, המידע האדום הוא מה שנשאר, והבלוק עצמו הוא בלוק 2. נעתיק אותו לדיסק, ונעתיק את השדות החדשים.

ה-Buffer Cache ה-Buffer Cache מוקצה עבור מידע מה-storage. כשמערכת ההפעלה רוצה לגשת לבלוק מסויים היא צריכה לבדוק אם הוא כבר ב-Buffer Cache. כיצד תעשה זאת? תעבור על כל הבלוקים? זה לא יעיל. פתרון אפשרי הוא מתן ערך Hash לכל בלוק, וקישור לבלוק לפי הערך.

כמו כן, עלינו לנקות את ה-buffer מבלוקים שלא צריך, כלומר, נשתמש ברשימה מקושרת ונזרוק את הבלוק לפי אלגוריתם ה-LRU. למה כאן אפשר להשתמש באלגוריתם? נבחין כי בכל קריאה מהזכרון אנו צריכים לחכות הרבה זמן, ולכן עדכון המצב של בלוק (Used/Unused) לא דורש תוספת זמן משמעותית.

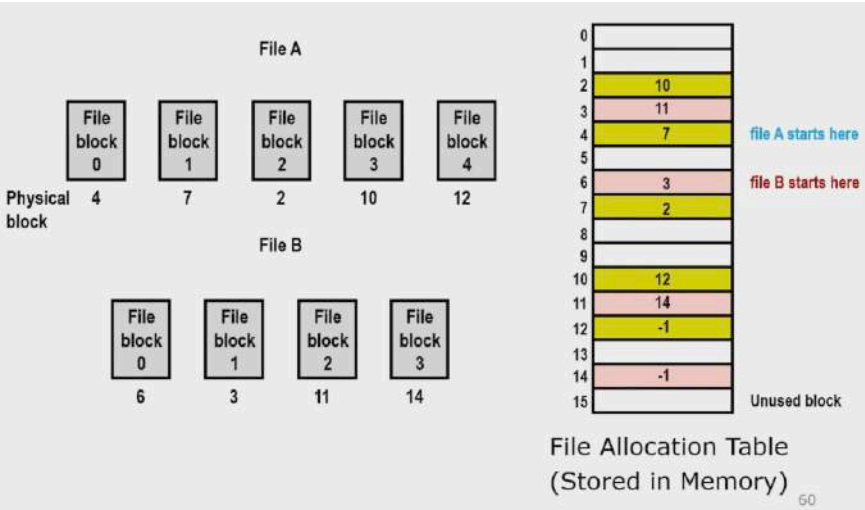
שאלה כיצד נמצא את הבלוקים על הדיסק?

יש שני מבנים נתונים שמתבקש להשתמש בהם. הראשון הוא רשימה מקושרת של הבלוקים $(MS-DOS)$. השני הוא שימוש באינדקס $(LINUX, UNIX)$ - בלוק מספר 1, 2 וכו'.



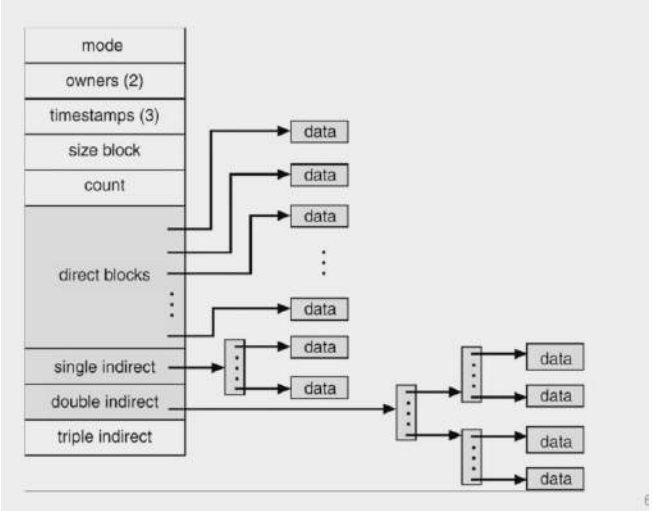
איור 60 : המחשה לשימוש ברשימה המקושרת. זה עובד, אבל נבחין כי כדי לגשת לבלוק בסוף הרשימה, צריך לעבור על כל הבלוקים לפניו **ולקרוא אותם**, שזה איטי מדי.

הפתרון לבעיה הוא שימוש ב-FAT. במקום שנצטרך לקרוא מהבלוק את הבלוק הבא, אנו נשמור פוינטר לטבלה שמכילה את המידע, ואז נצטרך לקרוא את המיקום של הבלוק הבא ממנה, כאינדקס.



איור 61: בכל פעם אנו קוראים את המידע מהטבלה, וחוסכים קריאה של זכרון מהדיסק. אמנם יתכן שנצטרך לעבור על כל הבלוקים, אך לא נבצע קריאות לדיסק. זה מאפשר גישה Random Access.

הדרך השנייה, שהיא שימוש באינדקסים, נעשית באופן הבא:



איור 62: נשמור רמה אחת של בלוקים באופן ישיר, כלומר נשמור את הפוינטרים באופן ישיר. הקבוצה השנייה תדרוש גישה לבלוק של פוינטרים. הקבוצה השלישית תהיה בעלת שתי רמות - בלוק של בלוקים של פוינטרים. ככה אפשר לשמור הרבה בלוקים בלי הרבה גישות, ובלי זכרון רב מדי לקובץ.
בפרט, 48k הראשונים נשמרים ב-direct
ה-4MB הראשונים נשמרים ב-single in − direct.
ה-4GB הראשונים נשמרים ב-double in − direct.
ה-4TB הראשונים נשמרים ב-triple in − direct.
כך כל בלוק יכול להמצא בלא יותר מ-3 גישות לדיסק. כאשר אנו שומרים ב-cache את הבלוק שמכיל את הפוינטרים כך שהמחיר של קריאת בלוק בפועל, לא דורשת הרבה קריאות מהדיסק.

הערה. הדבר הזה עובד כי התפלגות הקבצים מאוד מוטה. אמנם קבצים גדולים תופסים הרבה מקום, אך בפועל, הם תופסים כשליש משטח הדיסק, והשאר נתפס על ידי קבצים קטנים יותר.

הערה. בפועל 6% מהקבצים שוקלים פחות מ-60B. היות שה-inode כולל בתוכו 60B לפוינטרים עבור הבלוקים, אפשר להוסיף ביט שאם הוא ידלוק, נשמור את המידע של הקובץ בתוך הבתים האלה, במקום להשתמש בבלוקים. דבר זה שימושי מאוד כשמשתמשים ב-symbolic Links, שכן לא צריך בשביל זה בלוקים, אלא מעט בתים.

7.3.2 קבצים ממופים לזכרון

אחד החסרונות הבולטים במערכת הקבצים שתיארנו עד כה, הוא שהיא דורשת העתקה של בלוקים שלמים ל-buffer cache, ואז העתקה ממנו למקום הרלוונטי.

עקרונית, היה אפשר למפות קובץ לכתובת בזכרון באופן ישיר וכך למנוע את ההעתקה הנוספת. זאת אנו עושים על ידי שימוש ב-system call המכונה mmap, שממפה קובץ לפוינטר לזכרון. כך, כאשר נרצה לגשת לקובץ, כל שנצטרך לעשות הוא לגשת לכתובת הנתונה בפוינטר. במידה שהדפים הרלוונטים לא נמצאים בזכרון ה-ram נצטרך להעתיק אותם מהדיסק, אך מלבד זה, אין עוד העתקות.

היתרון כאן ה-direct access שמונע את ההעתקה מזכרון מערכת ההפעלה (buffer cache) למשתמש.

דבר זה שימושי מאוד כשאנו ניגשים לסמגנטים המכילים דפים רבים, שכן העתקות רבות נמנעות. כמו כן, עם סיום השימוש בסגמנט, נבצע unmmap, שימחק את המיפוי מהכתובת הוירטואלית לקובץ, כדי שיוכלו להשתמש בכתובת הנ"ל שוב.

הערה. סך הכל, מדובר בממשק נוסף לניהול נתוני הקבצים.

7.3.3 ניהול זכרון הדיסק

בעבר כדי לגשת לדיסק, מערכת ההפעלה הייתה צריכה להכיר את מבנה הדיסק - כמה משטחים יש, כמה מסלולים יש בכל משטח וכמה סקטורים בכל מסלול.

הממשק בין מערכת ההפעלה לדיסק, דרש לדעת איזה מסלול בדיסק, איזה סקטור במסלול, איזה ראש קורא אנו רוצים - יש שני ראשים, אחד מלמטה ואחד מלמעלה.

לכן, עם מבנה זה הגיוני לסדר את המידע כך שמידע של קובץ יהיה באותו מסלול, כולל ה-inode, כלומר, שלא יהיה רחוק מהמידע של הקובץ, ואז כשנרצה לגשת למידע, לא נצטרך להזיז את הראש הקורא יותר מדי. אחד החסרונות הבולטים בשיטה זו, הוא שהיא לא עמידה בפני סקטורים שהפסיקו לעבוד. בעבר, חברות היו מספקות לצרכן בדיוק לאילו סקטורים אסור לגשת, וזה "מלוכלך".

כיום זה לא המצב, לדיסק יש מחשב משלו, שכולל זכרון לא נדיף, והקריאה ממנו מתבצעת על ידי קריאה של מסלול שלם והעתקתו לזכרון הנדיף, מתוך תקווה שאם נרצה עוד בלוק, הוא יהיה באותו מסלול. זה חוסך את סיפוק המידע של איזה סקטור אנו רוצים. בדיסק מודרני זה, ה-controller שלו יודע בדיוק אילו דיסקים "דפוקים", והוא מספק ממשק של רצפים של בלוקים בלי להטריד את מערכת ההפעלה.

דבר נוסף הוא תזמון הבקשות על הדיסק. על פניו, היינו רוצים לתזמן את הבקשות כך שיתבצעו כמה שפחות גישות לדיסק, ולכן פותחו אלגוריתמים ייעודיים לכך כמו $\text{SSTF}, \text{FIFO}, \text{Scan}$, אך היום זה לא רלוונטי, כי האחריות עברה ל-controller ממערכת ההפעלה, והוא יודע לעשות זאת בעצמו.

7.4 פתיחה של קבצים

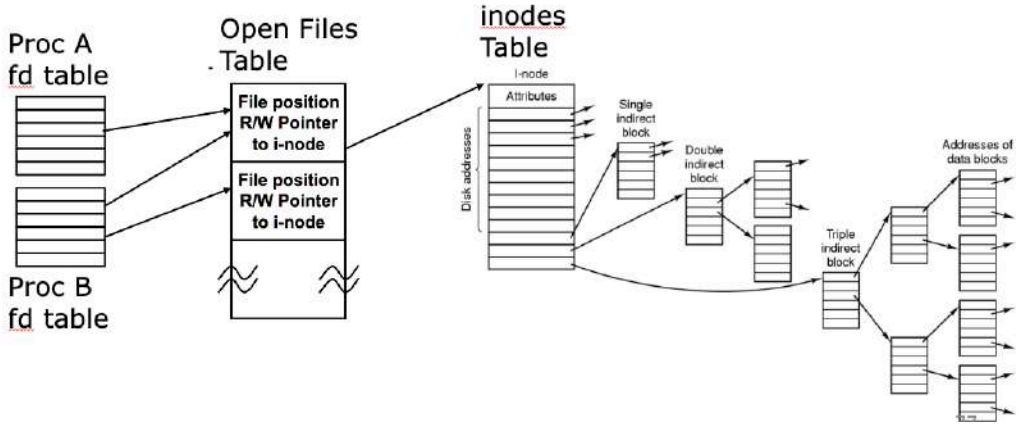
עד כה עסקנו במודל מאוד מופשט, בו יש טבלת קבצים פתוחים אחת, שממנה ניגשים ל-inodes. במודל זה, עולה השאלה, מה קורה כאשר fd מכניס כניסה אחרת בטבלה, למשל, מספר ראנדומלי כלשהו. לפי המודל, אנו ניגש ישירות לטבלת הקבצים הפתוחים וניגש ל-inode השמור בטבלה, אם יש שם, אם לא, נגריל מספר אחר. הקובץ שמתאים ל-inode כנראה נוצר על ידי תהליך אחר, והעובדה שניגשנו אליו, מהווה חולשת אבטחה חמורה. כדי לטפל בזה, אנו מבצעים הפרדה של טבלת הקבצם הפתוחים לטבלות של תהליכים.

כמו כן, דבר נוסף שלא לגמרי ברור, הוא מה קורה כאשר שני תהליכים פותחים את אותו הקובץ. לכל אחד מהם יש offset שונה, ולא רצוי שתהליך אחד ידע את ה-offset של תהליך אחר, על כן, צריך לשמור בנפרד את ה-offset של כל תהליך. כדי לאפשר את זה, מערכת ההפעלה שומרת שלוש רמות של טבלות פנימיות:

- טבלת ה-inode, שהיא טבלת הקבצים הפתוחים, ומכילה כניסה אחת לכל היותר, עבור כל קובץ, שכן אין סיבה לשמור inode פעמיים.
- טבלת ה-$\text{system} - \text{wide}$. זו טבלה שמערכת ההפעלה מוסיפה כדי לאפשר גישות של תהליכים שונים לאותו הקובץ. הטבלה מכילה כניסה עבור כל קובץ שנפתח, כלומר, היא טבלה גלובלית, שנגישה לכל התהליכים, וכל פעם שתהליך פותח קובץ, ה-offset של הקובץ נשמר בטבלה, בתא מיועד, ויתכנו כמובן תאים השייכים לאותו הקובץ.
- טבלת ה-$\text{pre} - \text{process}$. טבלה זו שמורה ב-PCB, וכוללת את רשימת הקבצים הפתוחים של התהליך, שזה אומר - מיפוי של fd לקובץ. האינדקסים של הטבלה הם ה-fd שנקראים גם ה-handle. למשל, $0 = \text{stdin}, 1 = \text{stdout}, 2 = \text{stderr}$.

  ◁ את שתי הטבלות האחרונות לעיתים מאחסנים באותה טבלה ביחד.

הדבר נראה כך:

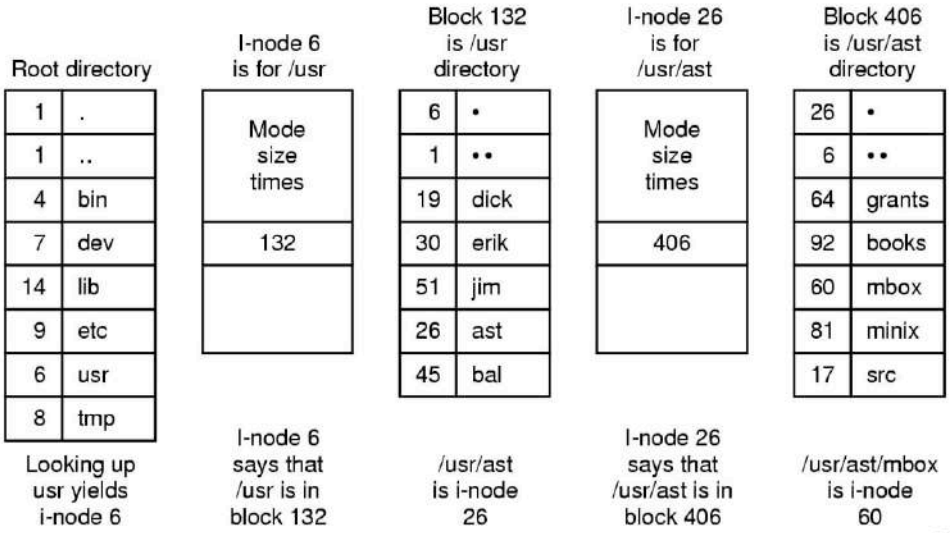


איור 63: תהליך A ניגש לקבצים הפתוחים של תהליך A בתוך טבלת הקבצים הפתוחים. זה נראה כאילו כל הקבצים נמצאים באותה הטבלה, אך זו הפשטה, בפועל, הטבלה של התהליך נמצאת ב-PCB.

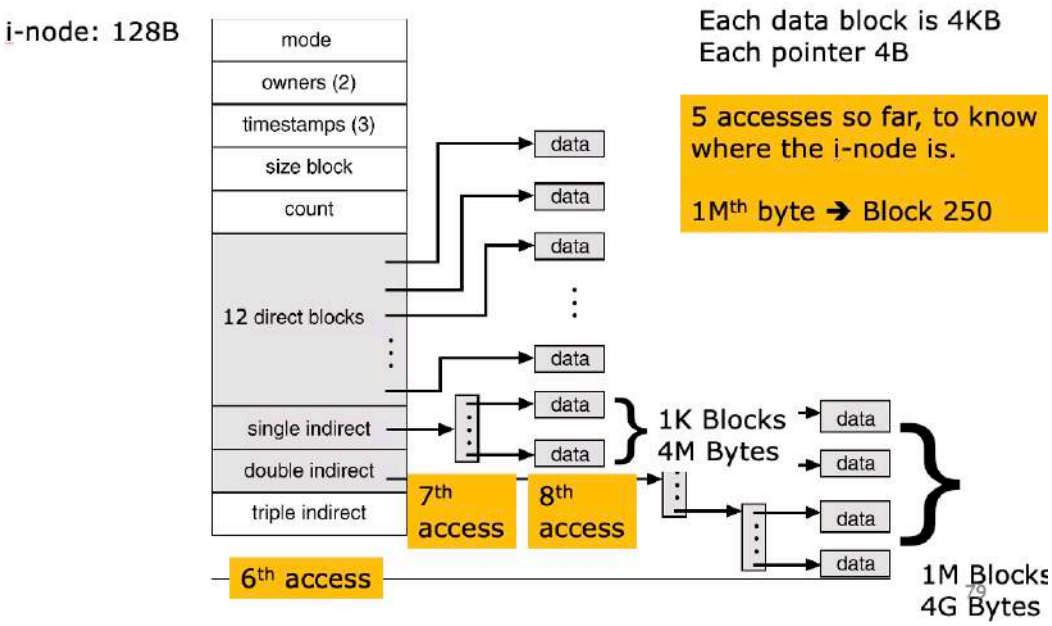
אם כך, מדוע תהליך B ניגש לקבצים של תהליך A? נניח כי A ביצע fork () ויצר את תהליך B, במקרה זה הקבצים הפתוחים של B יכילו את הקבצים הפתוחים של A, ולכן יוכל לגשת אליהם בטבלה.

נרצה לקבל המחשה לגישה לקובץ במערכת הקבצים, וכמה גישות לזכרון הדיסק נדרשות.

נניח כי מערכת הקבצים נראית כך:



איור 64 : אנו רוצים לגשת ל-"/usr/ast/mbox". תיקיית השורש פתוחה, וידוע המיקום שלה. לכן אנו מבצעים את הדבר הבא :
(1) : נחפש את המידע על usr. נמצא כי מתאים לו inode מספר 6. לכן הייתה לנו עד כה גישה אחת והיא קריאת המידע מהתיקייה. נקרא את המידע מ-$inode - 6$ ונראה שצריך לגשת לבלוק 132. ניגשנו לבלוק 132 וגילינו שמדובר בתיקייה. עד כה - 2 גישות.
(2) : נחפש את ast בתוך התיקייה ונמצא שמדובר ב-$inode - 26$. ניגש אליו ונגלה שהוא נמצא בבלוק 406. ניגש לבלוק ונראה שזו גם תיקייה. בשלב זה - שתי גישות (קריאת המידע מהתיקייה והגישה לבלוק).
(3) : בתוך התיקייה נחפש את mbox ונמצא שהוא ב-$inode - 60$. ניגש לבלוק, נראה שמדובר בקובץ. סך הכל 2 גישות.
בשלושת שלבים אלה היו לנו 5 גישות כדי לדעת איפה $inode - 60$ נמצא. הגישה ה-6 היא קריאת המידע מ-$inode - 60$



איור 65 : (4) : עלינו לקרוא את המידע מהקובץ, לפי השדות ב-$inode - 60$. אנו רוצים לגשת לבית מספר $1M^{th}$ לכן עלינו לגשת לבלוק 250.
אם נניח כי כל בלוק מכיל 4KB כניסות, זה אומר שהוא יכול להכיל $KB = 2^{10}$ פוינטרים. אם אנו מניחים כי יש 12 פוינטרים ישירים, ועוד שלושה לא ישירים, עלינו לגשת לפוינטר הרלוונטי שיהיה מסוג single, זו גישה מספר 7, שתוביל אותנו לפוינטר למידע הרלוונטי. ניגש אליו ונגיע למידע, זו גישה מספר 8.
סך הכל 8 גישות.

7.4.1 Mounting

דבר מסתורי שלא פירטנו עליו הוא גישה למערכת קבצים נוספת מלבד מערכת הקבצים הקיימת במחשב. למשל, נניח שאנו מחברים diskonkey למחשב. כיצד מערכת ההפעלה יודעת לגשת לקבצים בתוכו? לדבר זה קוראים mount, והיא system call.

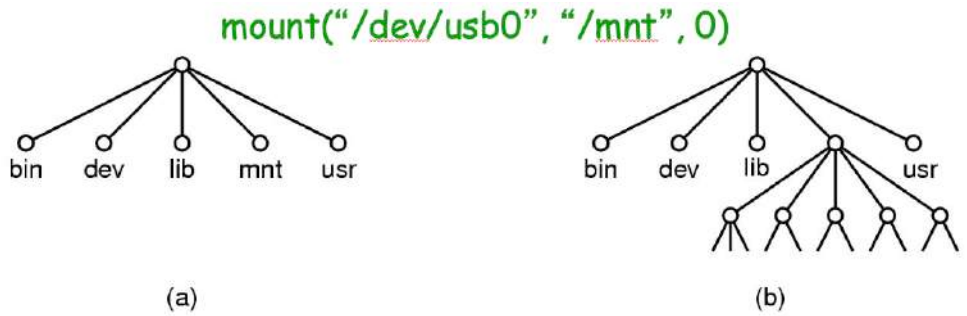
פקודה זו הופכת את מערכת הקבצים החדשה לתיקייה בתוך מערכת הקבצים הראשית, כך שכשניגש לתיקייה, ניגש למערכת הקבצים החדשה, משם

נוכל לקרוא מידע לפי אופן פעולתה.

למשל, הפקודה

$$\text{mount}\ (\text{"/dev/usb0"}, \text{"/mnt"}, 0)$$

תהפוך את מערכת הקבצים של usb0 לתיקייה mnt/ תחת תיקיית השורש:



איור 66: בעוד בפועל הגישה ל-usb0 מתבצעת לפי הנתיב "/dev/usb0" כלומר תחת התיקייה "dev", הפקודה אפשרה גישה לרכיב דרך תקיית השורש, כאילו שמדובר בתיקייה בפני עצמה.

7.4.2 Network File System (NFS)

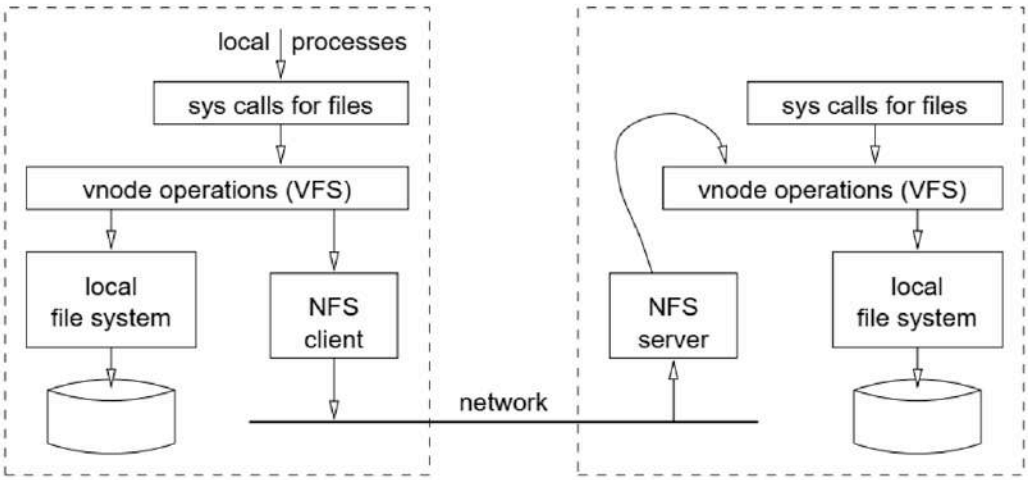
שאלה הכל טוב ויפה, אך מה קורה כאשר מערכת הקבצים נמצאת על מחשב אחר, כלומר לא על המחשב שלנו, כיצד נוכל לבצע mount?

התרחיש שתואר מאוד נפוץ. למשל במחשבי האוניברסיטה, כשאנו פותחים אותם, יש לנו גישה למערכת הקבצים, אך בפועל, המערכת שמורה במחשבים בקומה -2 בבניין B. למרות זאת, הקבצים נשמרים גם לאחר שסוגרים את המחשב. אם כך, מה קורה כאן?

אנו משתמשים ב-NFS, שהיא מערכת קבצים רשתית. הדבר מתבטא בכך שאנו מוסיפים עוד שכבה בעת גישה לקובץ והיא שכבת ה-vnode. שכבה זו תזהה האם הגישה היא גישה למערכת קבצים רחוקה, או למערכת הקבצים הלוקאלית. אם זה גישה לוקאלית, נגיע למערכת הקבצים הרגילה שלנו, והכל יתרחש כמו שכבר ראינו.

אבל אם מדובר בגישה מרחוק, יווצר לקוח NFS שיתקשר עם שרת NFS במחשב המרוחק. השרת יקבל את הבקשה ויעביר אותה ל-vnode במחשב המרוחק, שם הוא יבחין שמדובר בבקשה **לוקאלית**, שכן הגישה היא לקובץ בתוך מערכת הקבצים של המחשב המרוחק, מתוך רכיב במחשב **המרוחק**.

הסכמה הבאה מתמצתת את עיקרי פעולתה:



איור 67: ניתן לראות את העברת הבקשות דרך ה-Vnode לשרת. הפעולות שאנו מבצעים מתרגמות **להודעות** שנשלחות לשרת. השרת קורא את ההודעות ויודע לבצע את הפעולות בעצמו. הוא שולח חזרה את התוצאה, ואנחנו המשתמשים, רואים זאת כאילו שביצענו את הפעולות בעצמנו.
דבר זה יוצר וירטואליזיציה של מערכת קבצים מקומית, על אף שמדובר בגישה למערכת קבצים מרוחקת, מה שאנו קוראים לו "remote file system".

8 וירטואליזציה

המטרה בוירטואליזציה היא הפרדת התוכנה מהחומרה, כלומר, כתיבת קוד ללא התעסקות בפיסיקה של המחשב.

למשל, כאשר אנו ניגשים לאתר אינטרנט של האוניברסיטה, יש שרת שמקבל את הבקשות. האם אנו מודעים לקיומו הפיסי? לא בדיוק. כשאנו מבקשים כתובת ip כדי לגלוש באינטרנט, האם אנחנו צריכים ללכת לשרת ה-DHCP? גם לא. זו וירטואליזציה. הדבר תורם רבות לארגונים גדולים, שכן באמצעות וירטואליזציה אפשר להסתיר התעסקות פנימית עם השרת, מה שמחזק את השימוש בו. כמו כן, כיום אנו משתמשים בחישוב בענן (cloud computing), ווירטואליזציה מאפשרת לנו לגשת לשם בפשטות, ולחסוך קירור, חומרה, חשמל ותחזוקה.

עבור משתמשים, אפשר להריץ שתי מערכות הפעלה שונות על המחשב. דבר זה משונה, שכן, לא ברור מי שולטת בדיסק. אנו נפרט על כך בהמשך.

בפשטות, וירטואליזציה היא הדברים הבאים:

- הפרדת המימוש הפיסי מהמשתמש - מה שאנו מקבליפ, לא בהכרח קיים במציאות.
- גישה לא ישירה - הסרת המציאות המוגבלת. למשל, virtual memory מסתיר את הזכרון המוגבל של התהליך.
- מתן ממשקים זהים - תהליכים שונים צריכים לתקשר באותו האופן עם מערכת ההפעלה.

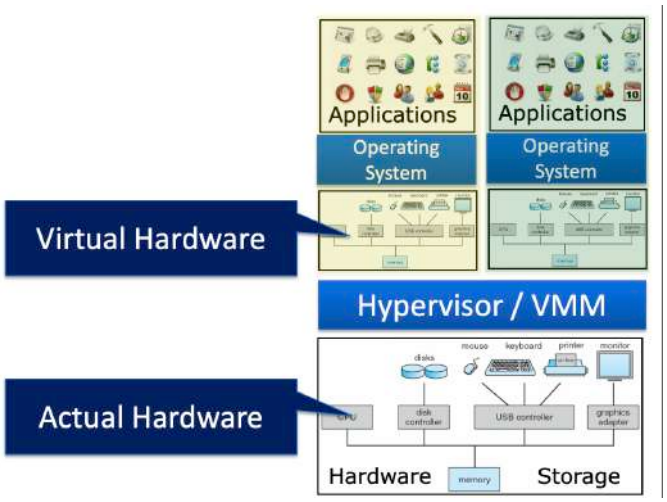
8.1 מכונה וירטואלית (Virtual Machine)

נבחין כי תהליכים הם למעשה וירטואליזציה של מערכת ההפעלה. יש להם זכרון, קבצים פתוחים, והם חושבים שכל המחשב שייך להם, למרות שהם יכולות לבצע פקודות מוגבלות ויש להם זכרון וירטואלי שאיננו פיסי. על כן, אפשר להתייחס להם כמכונה וירטואלית מופשטת (abstract), שכן רכיבי החומרה, אינם וירטואלים.

מכונות וירטואליות אמיתיות, הן קצת יותר מורכבות - הכל החומרה היא ויטואלית, כלומר מערכת ההפעלה יוצרת וירטואליזציה עבור המכונה. כל פקודות ה-CPU, כולל כאלה עם השראות מיוחדות, עוברות תהליך של וירטואליזציה דרך מערכת ההפעלה. כמו כן, הזכרון הפיסי ומיפוי הכתובות, כולל הרכיבים הפריפריאלים, כולם וירטואליים ועוברים דרך מערכת ההפעלה.

דבר זה מאפשר הרצה של מערכת הפעלה שלמה על המכונה הוירטואלית. עולה השאלה, כיצד בדיוק מתבצעת הוירטואליזציה?

אנו יודעים שמערכת ההפעלה מבצעת את הקישור של התוכנה עם החומרה. אך כיצד מתבצע הקישור של המכונה הוירטואלית לחומרה הפיסית? התשובה היא "שכבת ה-hypervisor":



איור 68: בעוד כל מכונה וירטואלית מכילה מערכת הפעלה, שיודעת לגשת לרכיבי חומרה באופן מסויים, כל גישה, עוברת לאחר מכן דרך שכבת ה-hypervisor/VMM. שכבה זו אחראית לפרש את הגישות לגישות לחומרה הפיסית. כך נוצרת חומרה וירטואלית עבור כל מערכת הפעלה.
דבר זה מאפשר Multiplexing - כלומר שימוש בכמה מערכות הפעלה על מכונות וירטואליות שונות. כמו כן, יש בידוד חזק בין המכונות, והשכבה מנהלת את הרכיבים הפיסיים.
בגדול, השכבה מבצעת פעולות שדומות מאוד למערכת ההפעלה.

אנו רוצים להבטיח שלוש תכונות בעת שימוש במכונה וירטואלית:

- שקילות - מכונה וירטואלית היא זהה למכונה הפיסית, כלומר כל האפליקציה שרצה בתוכה מתנהגת באופן דומה.
- בטיחות - מכונה וירטואלית מבודדת מהמכונה הפיסית וממכונות וירטואליות אחרות, כאילו שכל אחת רצה על מחשב אחר.
- ביצוע - יש שינויים מינוריים בהרצה על המכונה הוירטואליציה בהשוואה להרצה על המכונה הפיסית, כלומר אין הבדלי זמן ריצה משמעותיים.

בפרט, אם הצלחנו להבטיח את שלוש התכונות הנ"ל, נוכל לבצע וירטואליזציית multiplexing,

והיא יצירת כמה מכשירים וירטואליים על מכשיר פיסי אחד. הזכרון הוירטואלי יתמוך בכמה מרחבי כתובות שונים, במרחב כתובות פיסי אחד. נבצע חלוקה של הדיסק - כאילו שדיסק יחיד הוא כמה דיסקים, שכל אחד מספק זכרון בלתי תלוי באחרים.

וירטואליזציה נוספת היא וירטואליזציה באמצעות aggregation,

והיא יצירה של רכיב וירטואלי מתוך רכיבים פיסיים שונים. למשל, במקום שיהיו לנו עשרות שרתים לשירותים שונים, יהיה לנו שרת אחד גדול שיבצע את כלל השירותים, באמצעות וירטואליזציה.

רקע היסטורי

ההיסטוריה של הוירטואליזציה מתחלקת לשלושה שלבים עיקריים:

1. $1960s - 70s$ - הוירטואליזציה התבצעה על mainframes כלומר על מרכזי מידע רבים, למשל מחשב ענק שהיה אחראי לשירותים רבים. למשל, IBM אפשרה להשתמש ב-VM בתוכו וב-CMS שהוותה את מערכת ההפעלה בתוך המערכת עצמה.
2. $1980s - 90s$ - מעבר למחשבים אישיים, ויצירתה של חומרה שאיננה וירטואית.
3. $2000s$ - גילוי מחדש של יתרונות הוירטואליזציה - שימוש נרחב בחישוב ענן, ותמיכה חדשה בחומרה.

חישוב הענן שהפך לפופולרי בשנים האחרונות, מאפשר לשכור מכונות וירטואליות בשרתים פיסיים רחוקים, המבודדים היטב ממכונות אחרות בשרתים. דבר זה מאפשר למנוע עיסוק במכשיר הפיסי עצמו, ותחזוקה.

בנוסף, וירטואליזציה מאפשרת Server Consolidation - בעוד שרת בדרך כלל מריץ אפליקציה אחת עיקרית, נוכל לאפשר לרכיב פיסי אחד גדול (שרת חזק) להריץ כמה שירותים שונים, באמצעות וירטואליזציה. דבר זה חוסך בחומרה ואנרגיה, ומאפשר לאחד רכיבים פיסיים רבים לרכיב אחד. הדבר אפשרי כמובן, כיוון שרוב הזמן השרתים לא עושים יותר מדי, ולכן איחודם לשרת אחר, לא יוצר עומס על השרת הגדול. מלבד זאת, כאשר שרת מקבל בקשות רבות לאותו שירות, אפשר להפנות מכונות וירטואליות נוספות לשירות זה, וכך להוריד עומס.

זאת ועוד, בדיקות שבוצעו מראות ששימוש במכונות וירטואליות מרובות טוב יותר משימוש במכונה אחת בגדלים שונים.

הערה. למכונה וירטואלית שימושים נוספים. למשל, כאשר אנו מפתחים תוכנה והיא קורסת, יש לנו את מערכת ההפעלה שתדאג לטפל בזה. אבל מה קורה אם אנו מפתחים מערכת הפעלה ויש בה באג שגורם לקריסה? טיפול נאיבי יהיה טעינה של מערכת ההפעלה למחשב חדש, לאחר שקימפלנו אותה דרך מחשב אחר, ולאחר קריסה נקמפל גרסא נוספת במחשב אחר ונריץ שוב במחשב שבה היא קרסה. זה תהליך מאוד איטי. אם נשתמש ב-VM נוכל לטעון אותה, ואם היא תקרוס, ה-VM יקרוס, אבל מערכת ההפעלה שלנו תעמוד בזה, שכן מבחינתה מדובר בתהליך שרץ.

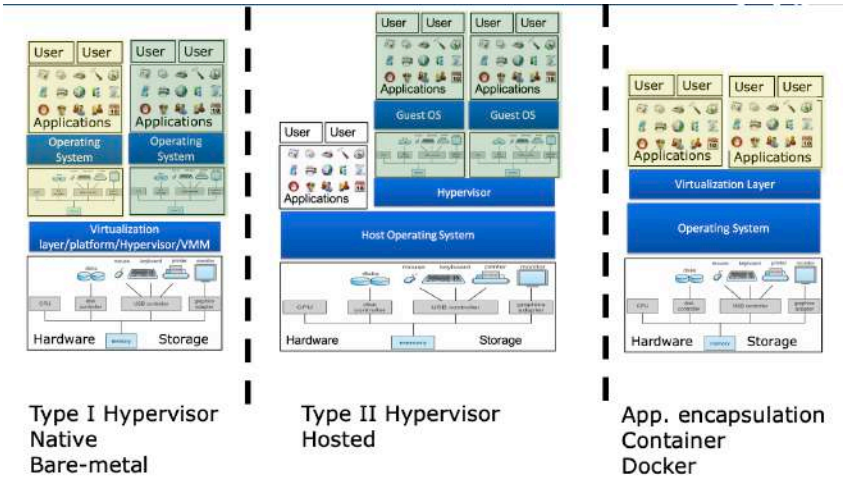
שימוש נוסף הוא הרצה של תוכנות ישנות. למשל, ATARI. היום אין לו תמיכה במערכות ההפעלה, אך על ידי התקנה של מערכת הפעלה ישנה על ה-VM נוכל להשתמש בו. כמובן שיהיה הרבה Overhead.

8.2 Hypervisors

כשאנו משתמשים ב-VM אנו לא צריכים להסתבך עם התקנה, יש לנו מכונה וירטואלית שלמה בקובץ אחד (מה שאנו מכנים אנקפסולציה)! שכולל מערכת הפעלה, אפליקציות, מידע וזכרון. כמו כן, המצב הנוכחי ב-VM נשמר ואנו חוזרים אליו כשמפעילים מחדש את ה-VM, מה שמאפשר גם גיבוי למידע. משהו אחד לא ברור, והוא, כיצד מתבצעת ההמרה בין מה שה-VM רואה לבין מה שהחומרה הפיסית מקבלת בסופו של דבר.

דבר זה מוביל אותנו לסוגים שונים של Hypervisors שמבצעים את הוירטואליזציה.

1. Hypervisor מקומי (OS) - נניח שאנו מריצים מערכות הפעלה שונות על אותו המכשיר, כן, **לא על** VM. תהיה לנו שכבת Hypervisor שתדאג לתרגם גישות לחומרה הוירטואלית לגישות לחומרה הפיסית. שכבה זו תדאג גם לחלק את הדיסק לפי מערכות ההפעלה השונות, ולדאוג שכל מערכת לא מודעת לקיום של האחרת.
2. Hypervisor של מערכת הפעלה ראשית (Guest OS) - נניח שאנו מריצים מערכות הפעלה שונות על VM, תחת מערכת ההפעלה הראשית. כל גישה לחומרה הוירטואלית ב-VM תעבור דרך שכבת ה-Hypervisor שתדאג לתרגם את הבקשה עבור מערכת ההפעלה **הראשית**, שתעביר את המידע לחומרה.
3. Virtualization Layer (Container) - נניח שאנו רוצים לגשת לאפליקציות שתומכות במערכות הפעלה אחרות, אנו יכולים להשתמש ב-Container שיכיל את האפליקציה, וכל הרצה שלה, תעבור דרך שכבת הוירטואליזציה, שתתרגם את המידע למערכת הפעלה שתוכל להעביר אותה לחומרה. כאן אין חומרה וירטואלית, בניגוד לשני הסוגים הקודמים.



איור 69 : כאן ניתן לראות המחשה ל-Hypervisors השונים.
קל לראון שרכיב מסוג 2 מונע שכפול של מערכת ההפעלה, ומשתמש במערכת ההפעלה הראשית מתחת לפני השטח, בעוד רכיב מסוג 1, משכפל מערכות הפעלה רבות.

למעשה, ה-Hypervisor הוא כמו מערכת הפעלה, ועבורו ה-VMs הם כמו תהליכים שונים, שהוא שולט עליהם - כיצד לתזמן אותם, כיצד להקצות זכרון, וכיצד לספק להם רכיבי I/O.

אבל כחלק מהוירטואליזציה, מערכות ההפעלה בתוך ה-VM חושבות שהן רצות ישירות על החומרה, ושהן שולטות בהכל, ולכן על ה-Hypervisor ל-"השלות" אותן. הוא עושה בשילוב של השיטות הבאות :

1. Trap and Emulate - הדבר העיקרי שמשתמשים בו ב-Hypervisors type 1.
2. Binary Translation.
3. Paravirtualization.
4. Hardware Assitance.

8.2.1 Trap And Emulate

הבעיה העיקרית בעת שימוש ב-VMs הוא העברת הבעלות ל-Hypervisor, כלומר, כיצד ניתן לוודא שכל בקשה לגשת לרכיב חומרה תעבור דרכו?

הפתרון הוא הדבר הבא.

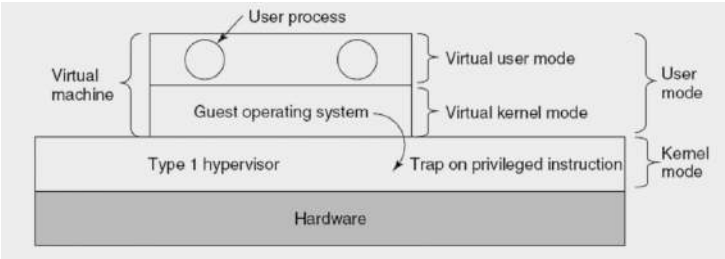
נריץ את כל רכיבי ה-VM ישירות על החומרה במצב User Mode. קראתם נכון, **ה-OS תרוץ במצב** User Mode.

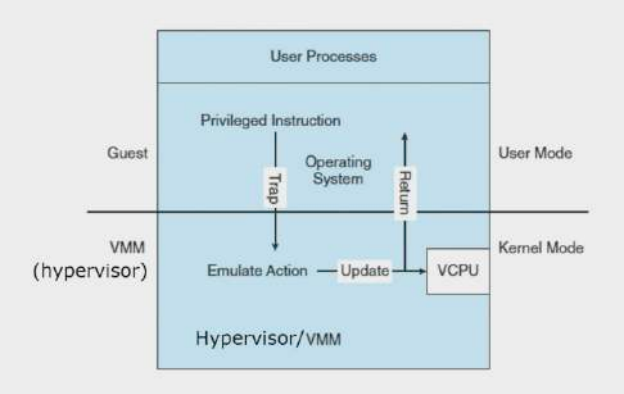
כך, כל עוד הפקודות שירוצו על המעבד, לא דורשות הרשאות מיוחדות, למשל, לחבר שני מספרים, לא תדרש גישה לחומרה, או לרכיבים "רגישים" ולא יהיה צורך ב-Hypervisor.

אבל, נניח שאחת האפליקציות רוצות לגשת לדיסק. הדבר דורש ביצוע system call, מה שדורש, טעינה של קוד של מערכת ההפעלה, שרץ במצב kernel mode. אבל, למערכת ההפעלה אין גישה כזו, אלא רק user mode ולכן בעת ביצוע הפקודה, תזרק חריגה של פקודה לא תקינה. מי שיטפל בחריגה, הוא ה-Hypervisor, שידאג למנף את החריגה, להרצה על רכיבי החומרה הפיסיים. כלומר, הוא יקבל את האחריות לבצע את ההרצה.

דבר זה הגיוני, כל עוד מספר ה-traps אינו גדול מדי. כמו כן, אנו מניחים כאן שכל גישה לרכיב רגיש, דורשת הרשאות גישה, אך זה לא בהכרח המצב, למשל בשנות ה-80 מחשבי ה-intel86 לא עבדו כך.

נקבל לכך המחשבה האיור הבא :





איור 70: כאשר מערכת ההפעלה ב-VM מנסה להריץ קוד ב-Kernel Mode, מתבצע trap לקוד של ה-Hypervisor, שמבצע את הפעולה על הרכיבים הפיסיים.
בפרט, צריך לשים לב שלכל VM יש VCPU, כי בפועל, צריך לשמור את המצב של ה-CPU עבור כל VM.

8.2.2 Dynamic Binary Translation

תרגום דינאמי של כתובות בינארי מתבצע כאשר אנו ניגשים לרכיב רגיש, שדורש הרשאות גישה USER, ולא של מערכת ההפעלה, כלומר, כשה-VM תרצה לגשת לחומרה, לא יתבצע trap ל-Hypervisor. לכן, יתבצע תרגום של הכתובת, כך שבכל גישה לרכיב רגיש, הכתובת תתורגם לכתובת הניגשת לרכיב, אך דורשת הרשאות גישה מסוג kernel, כדי שיתבצע trap.

השימוש הראשון בתרגום דינאמי זה, היה לאחר שהוכנסו אופטימיזציות למערכת ההפעלה, שאפשרו ל-system calls מסויימות לרוץ במצב User, כשהגיעה הוירטואליזציה, הדבר היה בעייתי במיוחד, ולכן נדרשו לבצע את התרגום.

8.2.3 Paravirtualization

נשנה את ה-OS כך שכל הקריאות שדורשות הרשאות גישה מיוחדות, ישתנו לקריאות ל-Hypervisor כלומר systemcalls $\Rightarrow$ Hypercalls.
זה קורה כיוון שהמכונה הוירטואלית לא משתפת פעולה עם ה-hypervisor, אלא קוראת (מנסה) ישירות למערכת ההפעלה.
לכן מעבירים את הקריאות ישירות ל-hypervisor. דבר זה מוריד את כמות ה-traps, כמות הפקודות הרגישות - אין את ה-exception שנובע מפקודה ללא הרשאות.

יחד עם זאת, צריך גישה לקוד המקור של מערכת ההפעלה, על מנת שתתאים ל-hypercalls. לכן, צריך לקמפל את הכל מחדש עבור כל מערכת הפעלה חדשה.

היות שאין כאן שינוי בחומרה, קל יותר לשנות את קוד המקור.

8.2.4 Hardware Assitance

היות שוירטואליזציה הפכה לנפוצה, התחילו להשתמש באופטימיזציות ברמת החומרה.

כלומר, נוכל להוסיף הרשאות חדשות, פקודות חדשות, ומבני נתונים חדשים ברמת החומרה. לא נפרט, רק נעיר ש-VMware היא החברה החלוצה בתחום.

8.2.5 Containers

קונטיינרים הם שלב ביניים בין תהליכים לבין VM. למשל, בעוד תהליכים חולקים את אותה מערכת קבצים, קונטיינרים לא חולקים. אך בעוד מכונות וירטואליות מבצעות וירטואליזציה לחומרה, קונטיינטרים לא מבצעים.

יש להם אפליקציות, ספריות וקבצי קונפיגורציה. הם יוצרים וירטואליזציה של מערכת ההפעלה, מה שאומר שלכל קונטיינר יש מערכת קבצים משל עצמו, משאבים למערכת ההפעלה, כאילו שהוא רץ לבד.

המוטיבציה מאחוריו, הוא התרחיש הבא. נניח שיצרנו קובץ בינארי להרצה במחשב עם מערכת הפעלה מסוג $MAC - OS$ ואנו רוצים להריץ אותו על מחשב עם מערכת הפעלה מסוג Windows. הדבר לא יעבוד. מדוע? הקובץ מניח שמות של קבצים, מיקומים, ספריות קיימות ועוד, שלא בהכרח קיימים בווינדווס. לכן, על ידי התקנה של קונטיינר עם הרכיבים הרלוונטים, נוכל להריץ אותו על Windows.

נוכל להסתכל עליהם כאילו שהם אורזים תהליכים כדי שיוכלו לרוץ עם משאבי מערכת ההפעלה המתאימים להם. לדוגמא, משאבי Ubuntu עבור valgrind, שלא רץ על $Mac - OS$.

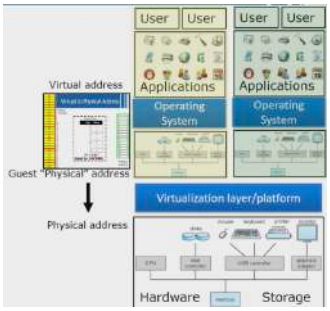
אריזה זו מוגדרת באמצעות שני גורמים:

- Namespace - עוטף משאב של מערכת גלובלית באופן אבסטרקטי כך שתהליך שרואה אותו תחת ה-namespace, חושב שיש לו משאב גלובלי מבודד משלו. למשל, כשאנו מבצעים mount אנו ניגשים לקבצי מערכת הקבצים כאילו שהם שלנו, למרות שבפועל הם שייכים למישהו אחר. כמו כן, שמות של תהליכים וגישה אליהם הם גם namespace עבור הקונטיינר, כלומר הוא רואה אך ורק תהליכים שרצים בתוכו, ולא תהליכים חיצוניים.
- cgroup - Controlled groupds - המשאבים נשלטים על ידי ה-cgroups, כלומר, כמה מהמשאב ניתן לקונטיינר.

8.3 וירטואליזציה לזכרון הוירטואלי

כאשר אנו משתמשים במכונה וירטואלית, היא רואה מרחב כתובות פיסי ומרחב כתובות וירטואלי. המרחב הפיסי שלה, הוא למעשה מרחב וירטואלי, רק שהיא לא יודעת את זה,

ועלינו לתרגם את הכתובות בסוף, לכתובת פיסית אמיתית. כשמדובר במכונה אחת, זה נשמע פשוט, אך כאשר יש מכונה, בתוך מכונה, בתוך מכונה... זה כבר מורכב יותר, שכן כל פעולה נוספת, מגדילה את זמן התרגום ופוגעת בביצועים.



איור 71: ניתן לראות שהזכרון הפיסי של המכונה הוירטואלית הוא למעשה זכרון וירטואלי

הפתרון הפשוט הוא שימוש ב- Shadow Page Tables. מערכת ההפעלה תשמור טבלות דפים שיכילו מידע על מה שהמכונה הוירטואלית צריכה, ומה שהמכונה הוירטואלית רואה בפועל.

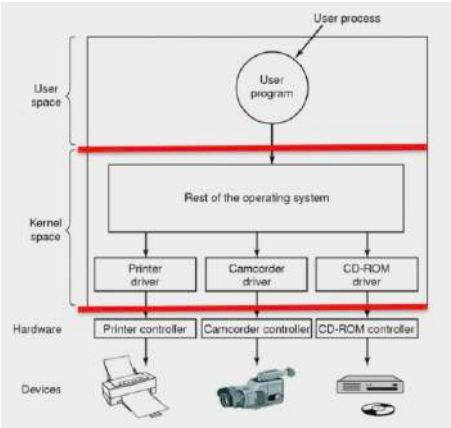
הטבלות יתמרנו בין נקודות המבט השונות וינסו להגביר את רמת הביצועים. הן מכונות "Shadow", כיוון שהם קיימות במערכת ההפעלה העליונה, ולא במכונה הוירטואלית, ומטרתן לספק מידע על משהו "שלא באמת קיים", ובכך ליצור וירטואליזציה. זה מבלבל, אבל לא נעמיק מעבר לכך.

הדבר דורש לתחזק את הטבלות כאשר המכונה הוירטואלית משנה אותן, שכן היא בטוחה שהיא שינתה אותן, אך בפועל, אלה לא הטבלות האמיתיות, שכן היא לא מודעת ל-shadow tables. בנוסף, צריך לדעת לגשת ל-TLB באופן מספיק חכם, שכן אנו עוברים בין מכונות וירטואליות, ולא בין תהליכים, מה שמוסיף מורכבות, היות שכל אחת רואה משהו אחר.

בכל הנוגע לתמיכה בחומרה, אפשר להשתמש בטבלות ברמת החומרה - nested paging, שזה שימוש ב-Page Tables ברמת החומרה, מה שמקטין את זמן הריצה, על אף שהוא מגדיל את כמות הגישות לזכרון - שכן במקור, הכל ממומש בתוכנה, והמעבר לתוכנה, בסוף יוצא איטי יותר.

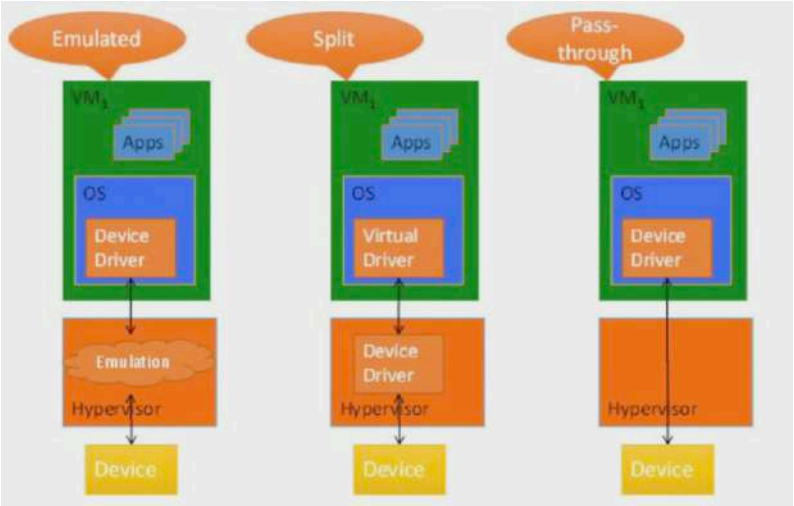
8.4 וירטואליזציה לרכיבי I/O

ניזכר כי בגישה לרכיבי I/O יש וירטואליזציה, שמונעת מאיתנו להתעסק בהפעלה הפנימית של הרכיבים:



איור 72: מערכת ההפעלת מתקשרת עם ה-Driver **בתוכנה** והוא יודע לתקשר עם ה-controller בחומרה.

וכפי שראינו יש כמה סוגי וירטואליזציה, וצריך לבחור במה להשתמש.



איור 73: אפשר להשתמש ב-type 1 ואז כל דרייבר נמצא מעל ה-hypervisor. זו גרסא מאוד פשוטה, שכן הדרייבר לא מודע לוירטואליזציה, וה-hypervisor תופס את כל הבקשות שלו ומבצע להן אמולציה בשביל הרכיב האמיתי.
אפשר להשתמש ב-type 2 ולחלק את הדרייבר לוירטואלי במערכת ההפעלה ולפיסי, בתוך ה-hypervisor.
אפשר במקרים מסויימים שהדרייבר יהיה שייך למכונה וירטואלית אחת ולא לאחרות, למשל, אם בתוך המחשב שלנו יש מכונה וירטואלית שאיננה אינטראקטיבית, היא לא צריכה גישה לעכבר ולמקלדת, ולכן צריך דרייבר רק במערכת ההפעלה הראשית.

9 רכיבי I/O

9.1 מבוא

קיימים רכיבי חומרה מסוגים שונים: תקשורת וזכרון, וכל אחד מהם משתייך לקטגוריה אחרת:

- Input בלבד: המקלדת, העכבר.
- Output בלבד: המדפסת, המסך.
- Input/Output: כרטיס הרשת, הדיסק, מסך מגע.

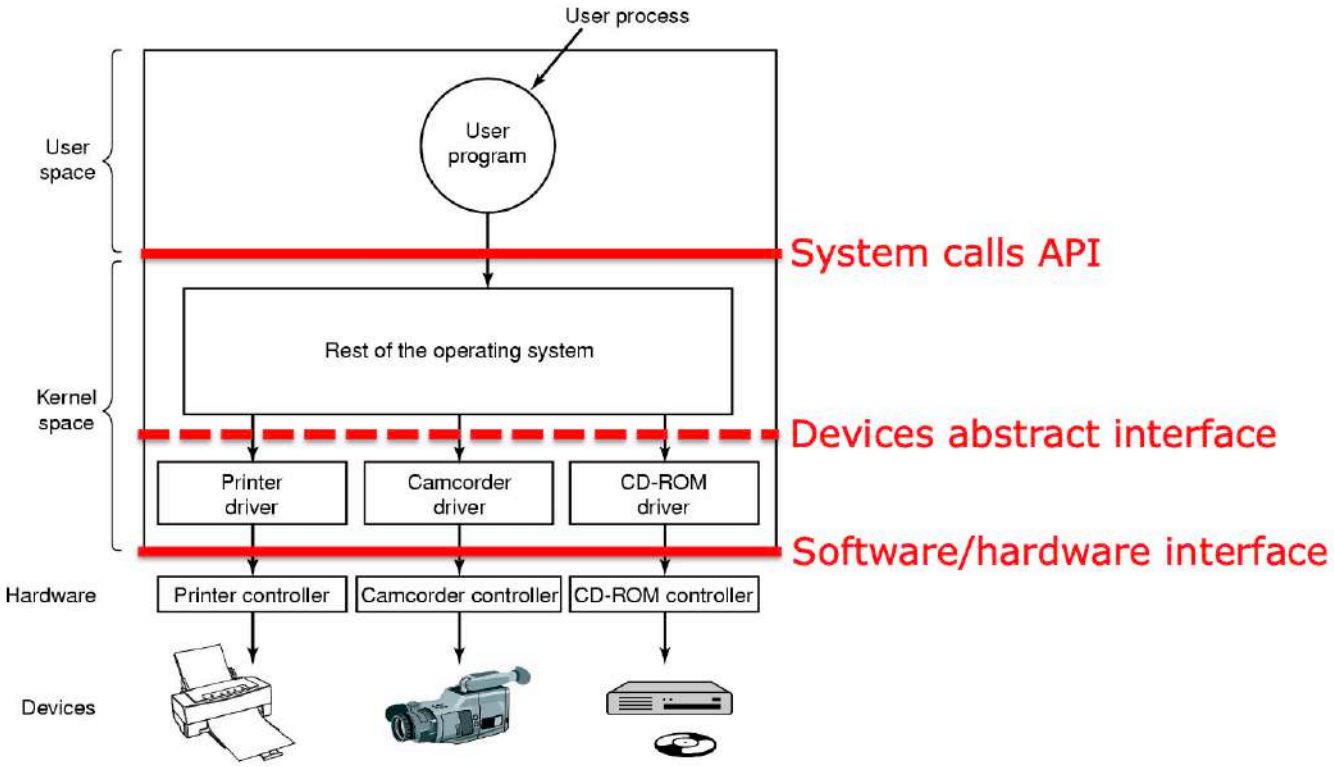
עד כה, דיברנו על הרכיבים מלמעלה, מנקודת מבטה של מערכת ההפעלה. עתה הגיע הזמן להבין בדיוק כיצד הם פועלים.
נתחיל בלשאול, כיצד מחברים רכיב I/O למחשב? כל רכיבי החומרה מחוברים לקונטרולר שלהם באמצעות bus. ה-Bus מורכב מהרבה חוטים, ובנוי לפי סטנדרט קבוע.
כל הרכיבים מתקשרים עם המחשב דרך אותו ה-Bus. לשימוש בו יש יתרונות:

- ורסטיליות:
  - ◁ אפשר לצרף רכיבים חדשים בקלות.
  - ◁ רכיבים שנמצאים במחשב אחד, יכולים לעבור למחשב אחר, כל עוד ה-Bus באותו סטנדרט.
- מחיר נמוך:
  - ◁ קבוצה קטנה של חוטים משותפת לכל הרכיבים באופנים שונים.

עלינו להבין מה תפקידי מערכת ההפעלה בכל הנוגע לרכיבי ה-I/O, שעל חלקם כבר דיברנו:

- הצגה לוגית של הרכיבים - הסתרת פרטי חומרה, וטיפול בשגיאות.
- שימוש יעיל - למשל, כאשר המעבד מריץ פקודה אחת, ה-I/O יריץ משהו אחר.
- שיתוף - הגנה כאשר I/O משותף, ותזמון הרכיבים. למשל, קבצים על הדיסק - תהליך ייגש רק לקבצים שלו. הדבר דורש גם טיפול בכרטיס הרשת.

נוכל להמחיש את היחס בין מערכת ההפעלה לרכיבי ה-I/O באיור הבא:



איור 74: בתוך ה-kernel נמצאים ה-drivers שמריצים קוד, שמתקשר עם ה-controllers.

9.2 Device Controllers

לכל יחידה יש את ה-controller שלה שמריץ קוד בחומרה, והוא לא חלק ממערכת ההפעלה, אלא מהחומרה.

יחידות ה-I/O מכילים רכיבים מכניים - כפור הפעלה, מקשים. יש גם רכיבים אלקטרוניים שהם מנהלים את כל התקשורת עם המחשב, אלה ה-controllers.

בעבר, היה חיבור מיועד לעכבר, למקלדת וכדומה, אבל היום הכל עובד דרך USB - זה עובד כיוון שהוא לא מעביר רק מידע, אלא גם חשמל, ולכן הרכיבים יכולים לקבל את החשמל שלהם ממנו, ולא מהמחשב.

בגדול, ה-controller הוא תחנת ביניים של תקשורת של מערכת ההפעלה עם ה-devices. עם השנים השתנו תפקידיו של ה-controller והוא קיבל אחריות גדולה יותר.

למשל, בעבר, מערכת ההפעלה הייתה צריכה לדעת כיצד לגשת לדיסק - איזה ראש להפעיל, איזה מסלול וסקטור בתוכו. כיום ה-disk controller עושה זאת לבד וחוסך ממנה. כלומר, הממשק בין מערכת ההפעלה ל-ctrl, נהיה "ידידותי" יותר בעוד ה-ctrl מבצע את פעולות ה-low level.

9.3 Device Drivers

מי שמדבר עם ה-controllers במערכת ההפעלה הוא ה-driver, וכל ctrl צריך driver שיהיה מסוגל להפעיל אותו. בגדול, ה-drivers נכתבים על ידי מי שמייצר את הרכיב, והוא מיוצר כך שמערכות הפעלה נפוצות יוכלו להשתמש בו.

ה-driver **הוא קוד** שרץ במצב **kernel mode**. זו נקודה רגישה, כיוון שמדובר בקוד שאנחנו לא באמת יודעים מה הוא עושה, ויש לו הרשאות מלאות.

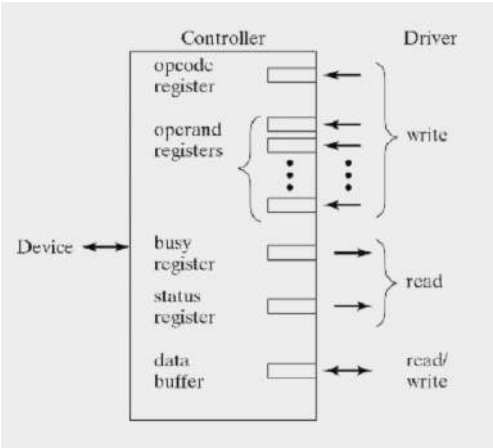
ל-driver יש ממשק משתמש שדומה מאוד למערכת הקבצים - write, open, close, seek, read, flush, כך שאפשר לקבל ממנו מידע ולשלוח לו מידע, וככל שהממשק אחיד יותר, כך התסבוכת נחסכת ממערכת ההפעלה.

ברמה הטכנית, ה-ctrl מקבל פקודות מה-driver, למשל להעביר תו למסך, לקרוא תו מהמשתמש, או לשלוח פקטה...

את העברת מידע זו, ה-driver וה-ctrl מבצעים באמצעות קבוצה של רג'יסטרים ייעודיים ב-ctrl. ה-driver כותב או קורא מהם באמצעות ה-bus, וה-ctrl מקבל מהם את המידע ויכול לתקשר עם הרכיב שלו.

את התקשורת של ה-driver עם ה-controller אפשר לעשות בשתי צורות:

- פקודה ישירה: יש פקודות אסמבלר שמיועדות לתקשורת עם רכיבים אלה.
- שימוש בפקודות קיימות: ניתן להרחיב את מרחב הכתובות כך שיכיל את הרג'יסטרים של ה-ctrl. ה-MMU יהיה מודע להרחבה זו, וכשאנו נכתוב מידע לכתובת מתאימה, הוא ידע שהיא מיועדת לאוגרים. לדוגמא, הפקודות lw, sw.

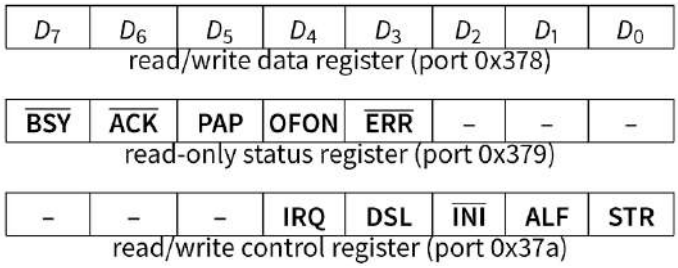


איור 75: ה-driver כותב ל-registers של ה-ctrl באמצעות ה-bus, וה-ctrl מצדו קורא את המידע מהם, או כותב אליהם בעצמו בשביל ה-driver, ועל פי זה, יכול לתקשר עם ה-device.

דוגמה. ה-LPT הוא חיבור מיוחד במחשב, שלא משתמשים בו כבר, אך היה נפוץ מאוד בעבר:

איור 76: חיבור ה-LUT במחשב והממשעות של כל כניסה בו

אותו חיבור, מאפשר לחבר רכיב חיצוני, וכל כניסה בחיבור מאפשרת קריאה או כתיבה לרג'יסטר ב-controller של הרכיב.



איור 77: האוגרים ברכיב

על מנת לתקשר עם רכיב שמחובר אליו, למשל, כדי לכתוב בית אחד לתוכו, נריץ את קטע הקוד הבא **ב-driver:**

```
void
sendbyte)uint8_t byte(
}
        /* Wait until BSY bit is 1. */
        while ) )inb )0x379( & 0x80( == 0(
                delay)(;
        /* Put the bute we wish to send on pins D7□0. */
        outb )0x378, byte(;
        /* Pulse STR )strobe( line to inform the printer
        * that a byte is avaiable */
        uint8_t ctrlval  = inb )0x37a(;
        outb )0x37a, ctrlval | 0x01(;
        delay)(;
        outb )0x37a, ctrlval(;
{
```

מה שאנו מבצעים כאן הוא קודם כל בדיקה שהרכיב פנוי, ואת זאת אנו בודקים על ידי בדיקת ביט ה-BUSY, שנמצא ברג'יסטר האמצעי באיור בפורט $0x379$. כדי לקרוא את המידע אנו משתמשים בפקודת האסמבלי inb, אך היא נותנת את כל ה-register, ולכן עלינו לבחור רק את הביט $\overline{\text{BSY}}$, שהוא הביט העליון ביותר, כלומר ביט מספר 8, לכן צריך להפעיל את המסכה $2^7 = 2^3 \cdot 2^4 = 0x80$. כל עוד הוא לא פנוי, מחכים חצי מיקרו שנייה.

לאחר מכן, אנו רוצים לשים את הבית ב-$D7 - 0$, כדי שנוכל להעביר אותו הלאה ל-ctrl. על כן, אנו כותבים אותו באמצעות הפקודה outb שכותבת לפורט $0x378$, שמכיל את הרגיסטר בשורה הראשונה.

כשכל המידע נמצא ב-ctrl registers עלינו ליידע את ה-printer שכתבנו byte. את זאת אנו עושים על ידי הדלקת ביט ה-STR בפורט 0x37a, לכן צריך לקרוא את כל המידע, לשנות את הביט, ולהעתיק חזרה. נקרא אותו עם הפקודה inb. נדליק את הביט הראשון על ידי הפעלת המסכה 0x01 | וכתיבה לפורט 0x37a. ככה כתבנו את כל המידע לרג'יסטר התחתון.

אבל, כאשר ה-printer יקח את הבית ויעשה איתו את מה שהוא צריך, עלינו לדאוג שהוא לא יראה שה-STR עדיין דולק, לכן, נחכה חצי מיקרו שנייה עם delay (), כך ניתן לו לסיים, ואז נחזיר את ה-0x37a פורט למצבו המקורי.

9.4 סוגי תקשורת (Universal Serial Bus)

עד לפני 20 שנה לכל רכיב היה חיבור יחודי - מקלדת, עכבר ומדפסת... אבל אז, כפי שציינו, הגיע ה-USB והחליף זאת עם סטנדרט יחיד, והוא הכיל פרוטוקול מיוחד לזיהוי הרכיב שהותאם אליו.

מאז התקשורת של הרכיב עם המעבד מחולקת לארבעה סוגים:

- Interrupt: הרכיב יוזם את התקשורת ומעביר כמויות קטנות של נתונים. למשל, מקלדת ועכבר.
- Bulk: העברת מידע בכמויות גדולות (Chunks) עם יחידה בסיסית של 64Byte, כמובן, עם קוד לתיקון שגיאות.
- Isochronous: העברת מידע כ-stream, כלומר מעבירים אותו בקצב קבוע, ללא תיקון שגיאות, כמו למשל, מיקרופון ורמקולים. מה שחשוב כאן הוא הזמן.
- Control: תקשורת בין הרכיבים ובין מערכת ההפעלה.

9.4.1 פעולת ה-USB

נניח שרכיב התחבר למחשב באמצעות USB. זהו חיבור משתנה - הוא יכול להתנתק בזמן שמערכת ההפעלה רצה.

ה-USB מזהה באיזה רכיב מדובר לפי הפרוטוקול שהוגדר מראש. סוג הרכיבים מתגלה. אם מדובר ב-isochronus/interrupt הם יאמרו כמה מידע הם רוצים לשלוח בכל פעם, ומערכת ההפעלה תסכים לחבר אותם, רק אם סך כל המידע שלהם, לא יתפוס יותר מ-90% מרוחב הפס. מעבר לסף זה, מערכת ההפעלה תסרב לקבל אותם.

העברת המידע מתבצעת ביחידות מידה של Frames = 1500Bytes, בתים מסויימים מוקצים ל-interrupts, isoch. ומה שנשאר מוקצה ל-bulk.

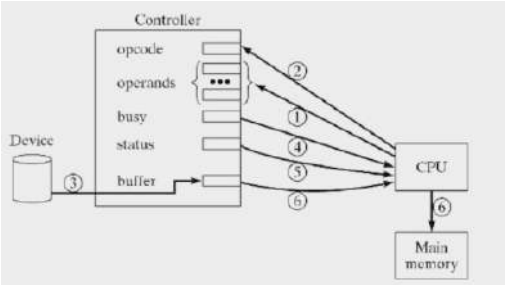
9.4.2 תקשורת Polling

ב-Polling המעבד אחראי על העברת המידע ל-buffer של ה-ctrl ולזהות מתי פעולת I/O הסתיימה. הוא עושה זאת על ידי בדיקה רציפה של אוגרי ה-ctrl.

מה שאומר שיש כאן busywait, כי הוא כל הזמן שואל ומחכה.

כדי לגרום לו להריץ פקודה, עלינו לכתוב קודם כל את הפרמטרים, שכן, אם שלחנו את הקוד, אנחנו לא יודעים אם נספיק לכתוב גם את הפרמטרים, הוא עלול להריץ מיד את הקוד.

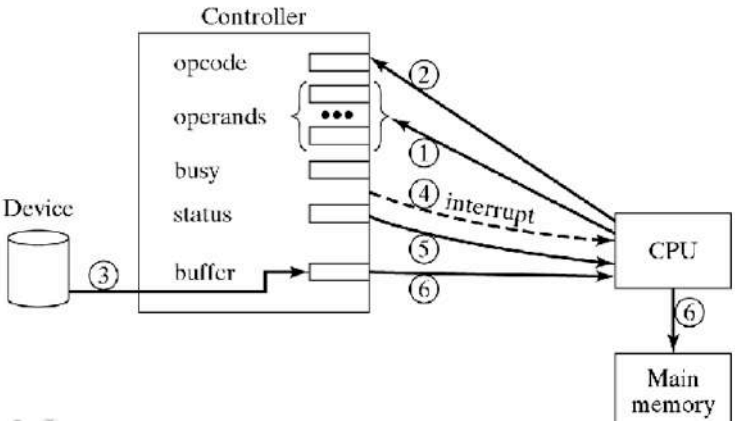
הפקודה תתבצעת המעבד יחכה לסיום, יבדוק שהכל בסדר ויקח את המידע.



איור 78: המחשה לפעולת ה-Polling. נבחין כי בשלב ההמתנה, יש כאן busy wait.

9.4.3 תקשורת Interrupts

גישת הפסיקות שונה, והיא שכאשר הרכיב מריץ את הפקודה, המעבד יכול לבצע דברים אחרים, וכאשר הרכיב יסיים הוא ישלח interrupt.



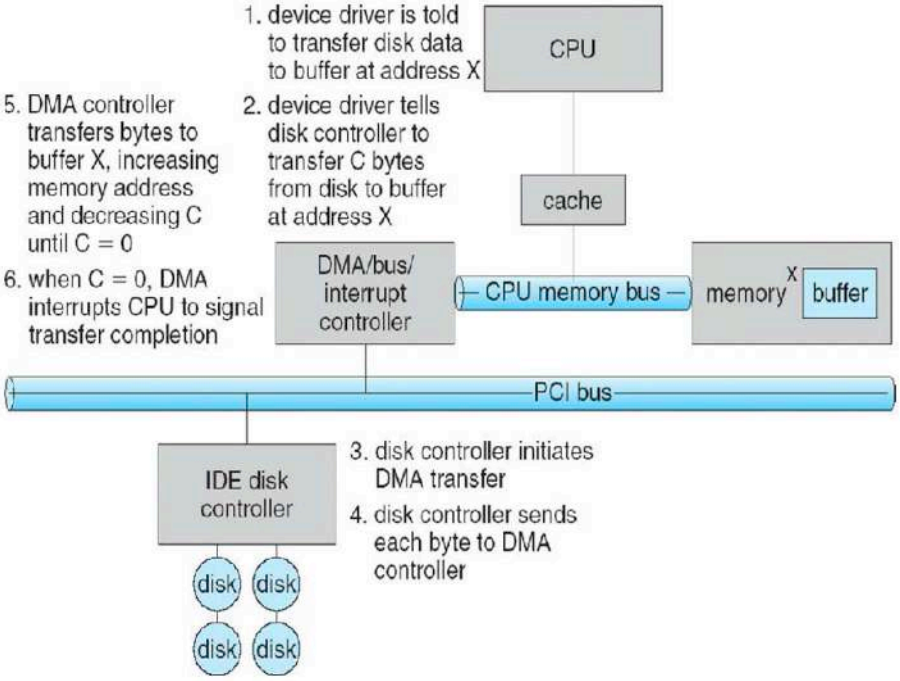
איור 79: כאשר הרכיב רץ ומסיים הוא שולח פסיקה למעבד כדי שידע הוא יכול לחזור לקחת את המידע.

כלומר ההבדל ממדיניות ה-Polling הוא שהמעבד לא ממשיך לבדוק האם הרכיב סיים, אלא הרכיב מודיע לו שהוא סיים, והמעבד חוזר לקחת את המידע, כאשר בזמן ביצוע הפקודה, המעבד מתפנה למשימות אחרות.

עולה השאלה, כיצד ה-driver יודע שרכיב סיים את ריצתו? וכיצד המידע עובר?

9.4.4 DMA (Direct Memory Access)

המעבד לא צריך להעביר את המידע, אלא רק להתחיל את הפעולה. ה-DMA ctrl יעביר את המידע ישירות לזכרון הראשי, ויזרוק interrupt כאשר ההעברה תסתיים.



איור 80 : המעבד מודיע לדרייבר.
ה-driver מבין שהוא צריך להעביר מידע מהדיסק ל-buffer בכתובת X,
לכן הוא מעביר ל-ctrl להעביר מספר בתים מסויים מהדיסק ל-buffer הנ"ל.
לאחר מכן ה-ctrl של הדיסק מתחיל העברת DMA,
כלומר הוא שולח בתים ל-DMA ctrl.
לאחר מכן ה-DMA ctrl מעביר את הבתים ל-buffer ומעדכן את כמות הבתים שנותרה להעברה.
כאשר נגמרו הבתים להעברה, ה-DMA שולח פסיקה למעבד כסיגנל לסיום ההעברה.

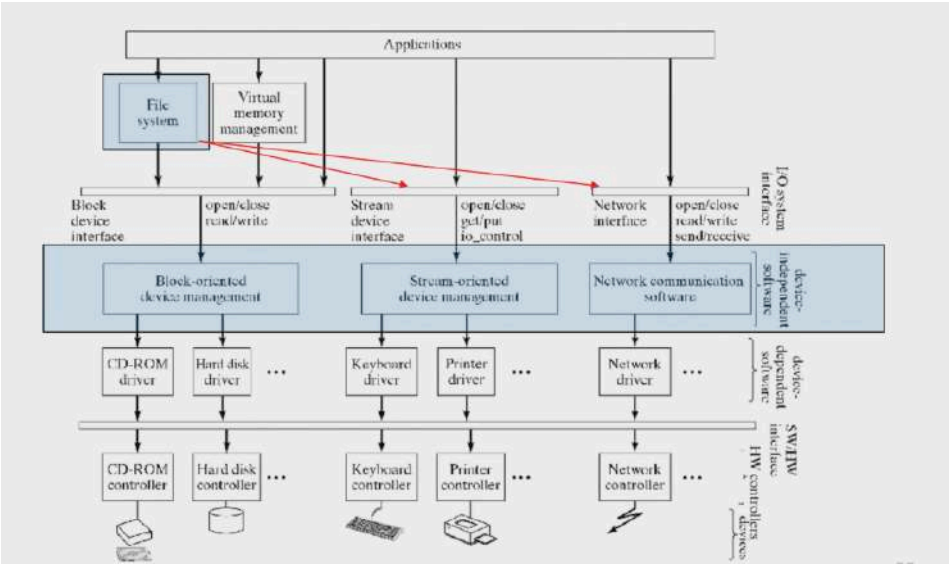
נבחין כי ה-DMA ctrl מתחרה עם ה-CPU על הגישות לזכרון, שכן הם עושים זאת דרך אותו bus.

בדרך כלל זו לא בעיה, כיוון שהמעבד ניגש לזכרון בתדירות גבוהה, אבל ב-cache, registers.

9.5 מודל היררכי

יש המון drivers, למעשה למעלה מחצי משורות הקוד של מערכת ההפעלה הן של drivers. רוב הזמן אנו משתמשים במעט מאוד רכיבים, אך מערכת ההפעלה צריכה להיות מסוגלת לגשת ל-ctrl המתאימים.

האופן שבו מערכת ההפעלה מסתכלת על ה-I/O הוא ההיררכיה הבאה :



איור 81 : אנו מגדירים לכל device לאיזה קבוצה הוא משתייך.
בפועל, מערכת הקבצים היא מודל לא רע בכלל, שכן פעולות פתיחה, קריאה, כתיבה וכו׳ דרושות גם לרכיבים הנ״ל ולכן המודל מאומץ גם עבור הרכיבים.
מה שאומר, שכל רכיב מיוצג על ידי קובץ בתיקייה dev (מלשון device), ואז אפשר לבצע read, write וכו׳, עם הרשאות גישה ומבנה דומה.

9.6 ניהול רכיבים

כדי לנהל את כלל הרכיבים צריך לבצע טיפול בשגיאות (כולל תיקון שגיאות), שמירת המידע ב-buffer/cache, ולתזמן את הרכיבים, ולשתף אותם.
למשל, נרצה שיהיה מספר פעולות מינימלי של seek בתוך ה-disk.

9.6.1 buffering

שימוש ב-buffering מאפשר גישה אסינכרונית לרכיב. שכן, הרכיב עלול להיות במצב busy אז נרצה שהנתונים שהמעבד רוצה לשלוח, יחכו לרכיב, וישמרו ב-buffer עד שיהיה פנוי.

זה גם מאפשר גישות במבנים שונים - סקטורים, בלוקים, גדלים שונים וכו׳.

ראינו זאת כשדיברנו על ה-buffer cache, ובסנכרון.

9.6.2 Error handling

קיימות תקלות רבות - תקלות תוכנה, תקלות בהעברת המידע. נדבר על זאת בהרחבה בקורס ברשתות.

בעקרון אנו מעבירים מידע, פלוס רכיב מידע נוסף שהוא פונקציה של המידע המקורי. כאשר היעד יקבל את המידע, הוא יפעיל את הפונקציה על המידע וישווה לרכיב המידע הנוסף.

למשל, תעודת זהות - ספרת הביקורת היא פונקציה של 8 הספרות הראשונות, ככה אפשר לזהות האם התעודה חוקית או לא.

10 Security

כשמדברים על אבטחה אנו צריכים להפריד אותה מאמינות. כאשר מדברים על אמינות, רוצים להבטיח שמערכת ההפעלה פועלת בצורה אמינה, למשל, נרצה למנוע deadlock. מערכות אמינות הן רובסטיות, כלומר אמידות בפני תקלות כאלה.

לעומת זאת, אבטחה מספקת בטיחות למערכת ההפעלה. למשל, הגנה מפני יריבים זדוניים משנסים לנצל חולשות קוד במערכת ההפעלה לטובת הרצה של קטעי קוד זדוניים במצב kernel mode. אבטחה מונעת מיריבים לפרוץ לתוך חשבונות של משתמשים, או לפרוץ ממש אל תוך המערכת ולהריץ מה שהם רוצים במצב kernel.

תופעה אחרת, שהיא data leakage, איננה מפגע אבטחה, אלא פשוט תכנון לא אמין של המערכת. למשל, אפליקציית Elector של הליכוד לא הסתירה את חשבונות המשתמשים שלהם, ולחיצה על המקש הימני של העכבר, כדי להסתכל על ה-source code, חשפה את כל המידע (6.5Milion). זה לא מפגע אבטחה, שכן אף אחד לא פרץ לשום דבר, אך זו דוגמא למשהו שצריך להימנע ממנו גם כן.

10.1 Authentication

מערכת ההפעלה מאפשרת למחשב שלנו לתת שירות ליותר ממשתמש אחד. כל משתמש מיוצג על ידי מזהה ייחודי משלו, ובעל הרשאות גישה שונות.

מלבד משתמשים כאלה, יש משתמש מיוחד שהוא ה-root (Administrator in windows), והוא יכול בעצם, לעשות הכל. עולה השאלה, כיצד ניתן להבטיח שמשתמש ניגש לחשבון שלו בלבד?

הפתרון שאנו מכירים הוא הכנסת שם משתמש וסיסמא לחשבון, וככה רק מי שידע את הסיסמא, יוכל להיכנס. מערכת ההפעלה תשווה את הסיסמא שהוכנסה לסיסמא ששמורה במערכת, ועל פי זה תדע אם להכניס את המשתמש או לא.

שמירת הסיסמאות במחשב, איננה ישירה, כלומר, אם הסיסמא היא Pass לא נשמור Pass, אלא נשמור את ה-Hash שלה. ככה, אם מישהו הגיע למאגר הסיסמאות במחשב, הוא עדיין לא יוכל לדעת מה הן. מה שאומר שעלינו לקבוע פונקציית Hash שתגבב את הסיסמאות עבורנו. יש לכך השלכה נוספת. כאשר מערכת ההפעלה בודקת את הסיסמא, אין לה למה להשוות, ולכן היא מחשבת את ה-Hash של הקלט ומשווה למה ששמור במחשב, אם פונקציית ה-Hash לא מספיק טובה, וערך ה-Hash ידוע, אפשר באמצעות מציאת התנגשות, לעלות על הסיסמא, למשל באמצעות הכלי Hashcat.

10.1.1 $\text{Brute} - \text{Force}$

תוקף שרוצה להיכנס לחשבון שלנו, או למחשב, יכול לבצע מתקפת $\text{Brute} - \text{Force}$ על הסיסמאות. כלומר, הוא ינסה את כל הסיסמאות האפשריות עד שיגיע לסיסמא הנכונה. כמובן, הוא יכתוב **תוכנה** שתעשה את זה, מה שאומר שהוא יוכל לבדוק המון סיסמאות.

הדרך בה הוא עובר על כל הסיסמאות במתקפה זו היא יצירת כל הקומבינציות האפשריות באלפאבת מסויים, שבדרך כלל יכלול את כל התווים הקריאים הקיימים (כדי למצוא אותם תוכלו לכתוב בפייתון (import string; print(string.printable)). ליצירת כל הקומבינציות יש אלמנט נוסף של אורך - ככל שאורך הסיסמא גדול יותר, כך יש קומבינציות, נניח שגודל האלפבאת הוא m ואורך הסיסמא הוא n, אז זה אומר שיש m^n אפשרויות לסיסמאות. מלבד זה שמדובר במספר עצום, כאשר n, m לא מאוד גדולים, נגיד $m = 20 = n$, מקבלים כמות סיסמאות שיקח יותר מדי זמן כדי לעבור עליה. לכן, אם הסיסמא מספיק ארוכה, והאלפאבת מספיק מגוון, המתקפה לא תצליח.

יחד עם זאת, אם הסיסמא קצרה, והאלפאבת מכיל רק אותיות, מאוד קל לעלות על הסיסמא. כמו כן, במידה שקובץ הסיסמאות השמורות במחשב דלפו, אפשר לנסות לתקוף את פונקציית ה-Hash, ולמצוא התנגשות.

נניח לרגע שאנחנו תוקף ואנחנו רוצים לקבל מידע רב על קורבן כלשהו. לאן נפרוץ? פרקטית, יהיה קל יותר לפרוץ למשאב אינטרנטי, נגיד, למייל שלו. אבל המייל שלו, מלבד הנגישות, יכול לאפשר לנו לשנות סיסמאות להרבה משאבים אחרים, שכן חברות רבות מאפשרות לשנות סיסמא באמצעות שליחת מייל. על כן, פריצה למייל היא מאוד מסוכנת ויכולה להוביל לגניבה אדירה של מידע.

10.1.2 ניחוש סיסמאות

למרות שבחירה של סיסמא מספיק ארוכה תצליח למנוע את המתקפה שתיארנו, בפועל, סיסמאות, גם אם הן ארוכות, כוללות תבנית מסויימת. למשל, הסיסמא תהיה באורך 10 תווים, עם מספר מאוד מצומצם של תווים, שייצגו מילים, למשל nameis1234, במקרה זה, אם התוקף מנסה להרכיב סיסמאות ממילים נפוצות, הוא יוכל לעלות על הסיסמא מאוד מהר.

כיום יש מאגרי סיסמאות נפוצות, ולכן על ידי הרכבה שלהן עם כל מיני מחרוזות אחרות, אפשר ביעילות לפרוץ סיסמא. מלבד זאת, הרבה סיסמאות מורכבות מצירופי מילים, ולכן אפשר לעבור על מילון ובאמצעות צירופים של מילים ממנו, לבנות סיסמאות. יתרה מכך, יש סטטיסטיקה שלמה לשימוש בסיסמאות, ולכן בחירה בסיסמאות על פי התפלגות, תניב פריצה מהירה.

השיטות הנ"ל מתבססות על האנטרופיה של הסיסמאות, במקום לבחור סיסמאות ראנדומיות אנשים בוחרים במכוון סיסמאות פשוטות שקל לזכור. לכן, כדי להקטין את האנטרופיה ובכך להקשות על הפורץ, עדיף להגריל את הסיסמא.

דרך אפשרית להתגבר על מתקפה זו, היא לשנות את היחידות. כלומר, במקום להתייחס לכל אות כיחידה, נתייחס לכל מילה כיחידה, ואז נוכל להגדיל מאוד את מרחב האפשרויות. כדי להקל על המשתמש לזכור את הסיסמאות אפשר להשתמש במנהל סיסמאות. הבעיה היא שגם הוא צריך סיסמא, אך אם היא ארוכה במיוחד, ואותה המשתמש מסוגל לזכור, אז זה חוסך לו הרבה סיסמאות אחרות.

10.1.3 מערכת ההפעלה

מערכת ההפעלה יכולה להתמודד עם מתקפות בדרכים רבות.

למשל, אם נוודא שפונקציית ה-Hash שאנחנו משתמשים בה איטית, זה יגביל את הכוח של התוקף, שכן מספר הניסיונות של יקטן כתלות בזמן, כדוגמא, במקום שבדיקה תיקח מיקרו-שנייה היא תיקח מילי-שנייה. אנחנו לא נרגיש את זה, אך התוקף כן.

נוכל להגן על סיסמאות דומות. נגיד, אם השתמשנו באותה סיסמא לכמה רכיבים שונים, והתוקף מצא את הסיסמא, תהיה לו שליטה על כל הרכיבים, ולכן נרצה למנוע זאת. הדרך שבה נעשה זאת היא באמצעות שימוש ב-Salt. במקום לכתוב לקובץ הסיסמאות רק את ההצפנה (Hash), נשרשר אליו רעש אקראי בגודל מסויים. הרעש אינו רק משורשר לשם, אלא גם לסיסמא המקורית, ועל התוצאה מופעל אלגוריתם ההצפנה, כך שגם אם הסיסמאות דומות, התוצאה הסופית תהיה שונה לחלוטין. זה מאוד שימושי כי בפועל יש הרבה חפיפה בין הסיסמאות.

דבר נוסף שנוכל לבצע כדי להסתיר את קובץ הסיסמאות, הוא הפיכתו לקובץ מיוחד. כל קובץ בעל שם ובאמצעותו אפשר להגיע ל-Inode שלו ולכן למידע שלו. מה אם נחליט שלקובץ אין שם, ונקבע לו רק inode נגיד Inode 13? מערכת ההפעלה תדע לגשת אליו, אך המשתמש לא יוכל! נעיר כי אם יש לנו גישה פיסית לדיסק של המחשב, וכתבנו מערכת הפעלה שמסוגלת לקרוא ממנו מידע, נוכל לשלוף את המידע ממנו. זה סיפור אחר, כדאי להבין שאם יש לנו גישה פיסית לרכיבים, עולם האבטחה משתנה.

לבסוף, דרך שבאמצעותה אפשר להגביל את כוחו של התוקף היא הגבלת מספר הניסיונות לגישה לחשבון כלשהו, נגיד 10 ניסיונות להכנסת סיסמא. או, להשתמש באמצעים ביומטריים.

10.2 הרשאות

לכל משתמש במחשב יש הרשאות גישה משלו, לכל תיקייה וקובץ יש הרשאות בהתאם למשתמש.

נבחין כי הרשאות מקנות פעולות, רוצה לומר, סך כל ההרשאות, מקנות למשתמש את כל מה שהוא יכול לעשות. אפשרויות פעולה (Capabilities) של משתמש אומרות מה הוא יכול לבצע על אובייקטים שונים. כלומר מדובר בדברים שונים.

עלינו להפריד בין שני סוגי משתמשים. סוג אחד הוא על מחשב, כלומר ב-dextop וסוג שני הוא ב-mobile, למשל בטלפון החכם, שם יש משתמש יחיד, והמון מכשירים. במחשב, יש הרבה משתמשים, מעט משאבים משותפים, מידע פרטי, מערכת הפעלה ששולטת על הכל, וקוד שאנחנו כתבנו.

במחשבים, יש לנו בקרה על גישות של תהליכים לאובייקטים שונים, כמו קבצים למשל. במובייל, יש לנו בקרה על גישות של אפליקציה למשאבים של מערכת ההפעלה, כמו קבצים, וקוד.[2]

כדי לשלוט בהרשאות הגישה, אפשר לרוץ כמשתמש רגיל עם sudo, מה שאומר שאנחנו משתמשים עם הרשאות גישה מיוחדות. אך אפשר לרוץ ממש כמו ה-owner אם נריץ את הפקודה setuid. זה מאוד שימושי כאשר אנחנו מריצים תוכנות שדורשות הרשאות מיוחדות, כמו ping שדורשת ריצה כ-root עבור שליחה של פקטות בקרה. או passwd שדורשת הרשאות כתיבה לקובץ הסיסמאות כדי לשנות סיסמא.

10.3 פרצות

פרצות אבטחה רבות מתבססות על תוכנה שלא בודקת את הקלט מהמשתמש. המשתמש עלול להכניס קוד זדוני לתוך המחשב, ועל ידי מניפולציה של הקלט, לגרום לתכנית להריץ אותו.

דבר זה יגרום לאפליקציה להריץ קוד שהיא לא אמורה להריץ, ואף יתן שליטה מלאה למשתמש על המכשיר.

דוגמה. למשל Log4j היא תוכנת Java לביצוע logging, היא שומרת מידע על ניסיונות כושלים לביצוע login, בוחרת איזה קוד שגיאה להעביר וכו׳.

היא אפשרה הרבה פעולות לביצוע, ואחת מהן הייתה לבקש את שם המשתמש מתיקייה מסויימת. הפרצה היא שלא נבדק איזה תיקייה הוכנסה, כלומר, לכל תיקייה שקיימת במערכת, היה אפשר לגשת (כל עוד ידעו מה השם שלה). מלבד זאת, התכנה שומרת ניסיונות כושלים להתחברות, מה שאומר שאם נכניס קוד זדוני, הוא יישמר במערכת. אם הקוד במערכת כל מה שנותר הוא להריץ אותו. על כן, תוקפים הקימו שרת זדוני, ששלח את הקוד, והקים תיקייה חיצונית ב-Web, כך שעם הרצת הקוד, המידע יועבר לתיקייה. הפרצה קיימת כבר מ-2013 אך התגלתה רק ב-2021. לא ידוע אם השתמשו בה, אך סביר מאוד שכן.

10.4 Malware

בעבר מערכת ההפעלה הייתה צריכה לדאוג שתהליך רץ לפי ההרשאות שלו. מזמן זה כבר לא המצב. צריך לוודא שהקלט תקין, שאין ניסיון פרצה, וכו׳.

מתקפת סוס טרויאני (Trojan Horse) היא מתקפה בה תוכנה חוקית מכילה "דלת אחורית" בעלת קוד זדוני.

וירוס, הוא תכנית "שזוהמה" בקוד זדוני, כלומר בבסיסה היא זדונית, לא כמו סוס טרויאני בהכרח. למעשה אחד הוירוסים הראשונים התגלה באוניברסיטה העברית בירושלים (כן, האוניברסיטה שלנו), ושני אנשים מוכשרים הצליחו לפענח מה הוא עשה, ולכתוב תוכנה שחוסמת אותו. זה האנטי וירוס הראשון! הומצא על ידי ישראלים מהעברית, הם הקימו את החברה BRM.

האנטי וירוס סורק את כל התוכנה ומנסה למצוא וירוסים ידועים או פעולות חשודות **לפני ריצה**. ה-Firewall חוסם כל קוד חשוד חיצוני שמגיע מחיבורים חיצוניים.

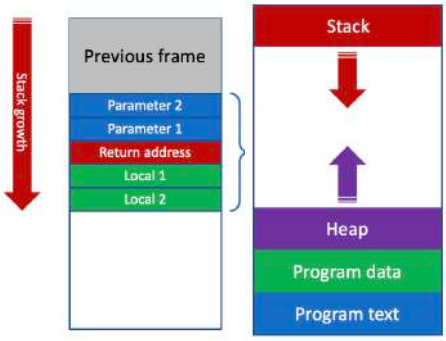
[2] התוכנה פגסוס, הסתמכה על קוד "לא בטוח" שרץ בכל טעינה של gif באפליקציה imessage. בין השאר, מדובר במשאב של מערכת ההפעלה שהאפליקציה משתמשת בו, לכן צריך בקרה.

10.5 Buffer Overflow

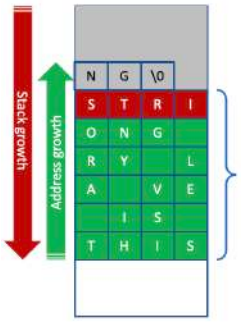
10.5.1 מתקפה

אחת הפרצות הנפוצות ביותר, היא ה-Buffer overflow, בה מעתיקים מידע ל-buffer, אך יוצאים מגבולותיו וממשיכים קדימה בזכרון. זהו באג בתוכנה, שניתן לניצול. שפות נמוכות הרי כמו $c, c++$ לא בודקות index errors כמו שפות גבוהות כדוגמת פייתון ו-java. כדי להתמודד עם זה אפשר לנסות ניתוח סטטי באמצעות הקומפיילר, ולהמנע משימוש מפונקציות לא בטוחות כמו strcpy, ולהשתמש בגרסאות בטוחות כמו strncpy, strlcpy.

תופעה שנובעת מפרצה זו היא ה-stack smashing, היציאה מהגבולות מגיעה עד לערך ה-ip ששמור בסטאק, ודורס אותו, כך שמה ששמור שם הוא כתובת למקום אחר בזכרון, שנוכל לקבוע להצביע לקוד זדוני שאנחנו הכנסנו.

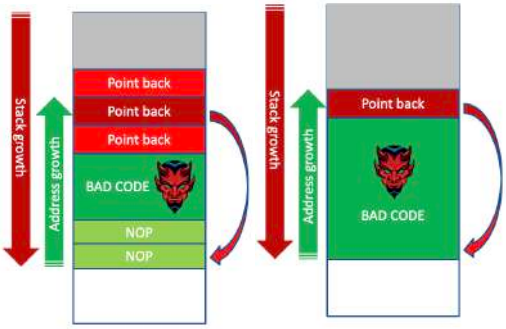


איור 82: מימין אפשר לראות את היחס בין הסגמנטים בכתובות **הוירטואליות**, הסטאק הוא הכי גבוה, מה שאומר שהוא גדל מטה. משמאל אפשר לראות את הערכים השמורים בסטאק בעת ביצוע של פונקציה מסויימת. נבחין כי אם נעלה מספיק בסטאק, כלומר נמשיך עם ה-overflow עד כתובת מספיק גבוה, נוכל להגיע ל-return address ולדרוס אותה.



איור 83: המחשה ל-buffer overflow, כתבנו מחרוזת ארוכה מדי לסטאק, ודרסנו את המידע בתאים האדומים, שהוא למעשה כתובת החזרה.

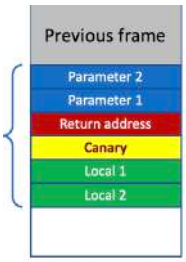
עולה השאלה, לאיפה נוכל להכניס את הקוד הזדוני. מאוד קל להכניס אותו לסטאק, שכן אנו יודעים את הכתובת שלו, אנו יכולים לכתוב לשם באמצעות ה-buffer, ואנחנו יכולים לגרום לכתובת החזרה להצביע למקום הרלוונטי בסטאק.



איור 84: מימין אפשר לראות המחשה להזרקת הקוד לסטאק ולדריסת כתובת החזרה כך שתצביע עליו.
יחד עם זאת, לעיתים אין וודאות מוחלטת בנוגע לכתובת הסטאק ולכתובת שצריך לחזור אליה, לכן, אפשר לרפד את הקוד בשורות nop שלא מבצעות כלום, ואז הקפיצה, בהסתברות גבוהה מגיעה לקוד הזדוני.

10.5.2 הגנה

נבחין כי המתקפה משנה את התוכן בסטאק, לכן נוכל להשתמש בערכי canary, כלומר, נשים ערך ראנדומי שיחצוץ בין כתובת החזרה למשתנים הלוקאליים, ולכן כשהתכנית תנסה לקפוץ לקוד הזדוני, היא קודם תבדוק את ערכי הקנארית, והיא תראה שהם גם השתנו, מה שיגרום לה לא לקפוץ.



איור 85 : המחשה לחוצץ. התוקף כמובן לא מודע לערכים שנמצאים שם.

דבר נוסף שאפשר לעשות הוא להקשות על התוקף לגשת לכתובות בזכרון. הרי, הוא ניגש למקומות בסטאק, ומנסה לקפוץ למקומות בסטאק. לכן, אפשר לבצע ראנדומיזציה של מרחב הכתובות, כך שכל כתובת בזכרון תעבור טרספורמציה לפני כן.אם אתם זוכרים, בתרגיל 2 התבקשנו להשתמש ב-address translator, זה בדיוק הראנדומיזציה. עשינו זאת כדי שהקוד יעבור, ויתאים למה שמערכת ההפעלה מצפה :

```
/*                                FROM EX2
   A translation is required when using an address of a variable.
   Use this as a black box in your code. */
address_t translate_address)address_t addr( }
    address_t ret;
    asm volatile)"xor□□□□%%fs:0x30,%0\n"
        "rol□□□□$0x11,%0\n"
                  : "=g" )ret(
                  : "0" )addr((;
    return ret;
{
```

אפשרות נוספת היא להוסיף בדיקה בעת ביצוע שורת קוד של המקור שלה. אם הוא הגיע מסגמנט הסטאק, לא נבצע אותה. כלומר נאפשר הרצה של שורות קוד אך ורק מסגמנט הטקסט. אממה, יש ספריות שלמות שכתובות בסגמנט הטקסט, והן כוללת את הפונקציות system, exec. מה שאומר שנוכל לקפוץ ישירות לפונקציות אלה אם נדע את הכתובות שלהן, ונקבל גישה ל-shell מה שיאפשר לנו לעשות מה שאנחנו רוצים. מתקפה זו נקראית $return - to - libc$ כיוון שאנחנו קופצים לפונקציות של libc.

10.5.3 Jail/Sandbox

הרבה קוד שאנחנו מקבלים מקורו באינטרנט. האם אפשר לסמוך עליו? ברור שלא. אז כיצד נוכל להריץ אותו?

דרך אחת היא להוריד את ההרשאות שלו, כמו שיש kernel mode/user mode, רק שההרשאות יהיו מאוד מינימליות כך שתכונות מהאינטרנט לא יוכלו לעשות יותר מדי.

דרך נוספת להתמודדות עם קוד זדוני, היא הרצת קוד מהאינטרנט בסביבה מוגבלת. למשל, ב-VM, אם אנו רוצים להיות מחמירים יתר על המידה. אך זה יקר מדי. אפשר לבנות סביבה שתדחה פעולות של מערכת ההפעלה שמגיעות מהמשתמש. נוכל לבצע זאת באמצעות וירטואליזציה, כמו שעשינו לקונטיינר.

דוגמה. מערכת ההפעלה Android נותנת לכל אפליקציה מזהה ייחודי, ומריצה אותה בתהליך אחר. כל הגישות מותנות לפי המזהה, יש הגנה על הזכרון בין התהליכים, ויש הגבלות על system calls.

הערה. בקורס בקריפטוגרפיה ואבטחת מידע עסקנו רבות במפגעי אבטחה, בפרט ב-buffer overflow, web בהרחבה. לכן, מי שמתעניין מוזמן לקרוא את הסיכום שבאתר.

זהו סוף הקורס. תודה על ההקשבה, אנו מקווים שהסיכום עזר לכן/ם וכל מה שנותר הוא לאחל לכולנו בהצלחה.

חלק I
תרגולים

הערה. לעיתים יש חפיפה בין התרגולים להרצאות, במקרה זה החומר לא יוצג בסיכום התרגול.

11 תרגול 1 - מבוא

הערה. בתרגול עשינו חזרה על שימוש וב-Valgrind, כיוון שראינו איך להשתמש בהם בקורסים קודמים, שהמודל שלהם עוד זמין, מצאנו לנכון לא להכניס זאת לסיכום. כמו כן, אפשר לקרוא על כך בהרחבה בגוגל.

נשאל שאלות, עליהן נענה במהלך הקורס:

- מה קורה כאשר אנחנו לוחצים על מקש במלקדת?
- מה קורה כאשר אנחנו קוראים או כותבים לקובץ?
- קריאת קובץ היא מאוד איטית, איך המחשב ממשיך להיות נגיש?
- איך אפשר לבצע כל כך הרבה דברים במקביל? מוסיקה, סריקת וירוסים, קריאת מסמך, על מעבד אחד?
- איך אפשר להריץ תכנית גדולה יותר מהזכרון של המחשב?
- מה זה segmentation fault?
- כיצד אפשר לזרז תכניות מחשב על ידי מערכות עם כמה מעבדים?
  - ◁ בעיה שעולה, היא מה קורה כאשר שתי תכניות כותבות לקובץ באותו הזמן?
- כיצד אפשר להריץ הרבה מערכות הפעלה על מחשב אחד? כלומר מעין תתי מחשבים על מחשב אחד.
- כיצד מחשבים מתקשרים אחד עם השני? כלומר שימוש ברשתות תקשורת.

הערה. נושאי הקורס הם אבני הבניין לתעשייה.

11.1 חזרה על מבנה המחשב

בקורסים קודמים למדנו מהו מחשב, וראינו שהוא מורכב ממעבד, זכרון, רגיסטרים ושבבי בקרה. הרגיסטרים הייחודיים הם:

- IR- שקול ל-A שראינו בנאנד, מחזיק את הזכרון של הפקודה במחשב.
- PC - IP ב-TASM, מצביע על הפקודה הבאה.
- SP - מצביע על המקום הבא במחסנית. במחשבים מודרניים, בשונה מנאנד, מדובר ברגיסטר בפני עצמו, שכן זה חוסך המון זמן ריצה.

המחשב מבצע פקודות מסוגים שונים:

- ניהול מידע - טעינת מידע על רגיסטר
- פעולות אריתמטיות - השוואה, bitwise, פעולות מתמטיות בסיסיות.
- פעולות בקרה - קפיצות מותנות ובלתי מותנות.
- פסיקות - יפורט בהמשך, לא ראינו בנאנד.

כל פקודה מתורגמת לקוד בינארי, אותו מריץ המעבד.
לכל פקודה יש מחזור חיים מוגדר היטב והוא:

1. שמירת הפקודה.
2. פענוח הפקודה וטעינת הרגיסטרים המתאימים.
3. פעולות ALU.
4. גישה לזכרון אם צריך.
5. כתיבה חזרה לרגיסטרים.

דוגמה. למשל הפקודה $\text{lw } \$1,\ 32\,(\$2)$ משמעותה $\text{Reg}[1] = \text{M}[\text{Reg}[2] + 32]$ - יש קריאה מהזכרון.

מסקנה. כיוון שלכל פקודה יש 5 שלבים, אין מניעה שיהיו כמה פקודות שונות משלבים שונים ב-CPU, במקביל.

11.2 היררכית הזכרון במחשב

1. רגיסטרים ב-CPU.
2. Cache - זמין ל-CPU והגישה אליו מהירה הרבה יותר מלזכרון. מידע שאנו ניגשים אליו הרבה במהלך התכנית ישמר בו.
3. זכרון - מחובר בפס ל-Cache, הגישה אליו איטית בהרבה, אך גודלו גדול יותר משמעותית - היום מכיל זכרון בסקלות של עשרות MB. הוא נמחק כאשר המחשב נכבה.
4. הדיסק IO/Deviced - מחובר בפס ל-Memory. היום מכיל זכרון בסקלות של מאות GB. הזכרון הזה לא נמחק אלא אם עשינו זאת במפורש.

11.3 GDB (GNU Debugger)

זה דיבגר שרץ על מערכות הפעלה מסוג UNIX עבור שפות תכנות רבות.
כדי להשתמש נריץ את הפקודה

$$\text{gcc} \ -\text{o myProg} \ -\mathbf{g} \ \text{test.c}$$
$$\text{g++} \ -\text{o myProg} \ -\mathbf{g} \ \text{test.cpp}$$

ואז נריץ

$$\text{gdb myProg}$$

אפשר להוסיף פקודות דיבגר סטנדרטיות:

- run - מתחיל את ריצת התכנית.
- break - עצירה בנקודה ספציפית (breakpoint).
- next - מריץ את הפקודה הבאה.
- step - ממשיך לפקודה הבאה.
- cont - ממשיך את ההרצה עד פקודה העצירה הבאה עד ה- (breakpoint).
- print *var* - מדפיס את הערך של var.
- watch *var* - לעקוב אחר ערך, על ידי הרצה של התכנית עד שהוא משתנה.

11.4 Clion - דברים שכדאי לזכור

כשמתמשים ב-Debugger שלו, אפשר להשתמש ב-breakpoints עם תנאי על משתנה, תנאי על הפונקציה הנוכחית, מבלי לבצע מספר רב של step.
ואף חלוקה ל-threads (רלוונטי לתרגילים מתקדמים יותר).

11.5 וירטואליזציה

וירטואליזציה זהו שם לתהליך יצירת מבנה וירטואלי של רכיב כלשהו שיכול להיות חומרה, מערכת הפעלה, זכרון ועוד.

מכונה וירטואלית VM, זו אמולציה למחשב. אפשר ליצור הרבה מכונות כאלה על המחשב שלנו, והמחיר הוא המשאבים שהן דורשות. כל אחת מהן מרגישה כאילו היא היחידה שמשתמשת במחשב.

למשל, נרצה להשתמש ב-linux על מחשב Mac, נשתמש ב-VM.

כיוון שהן דורשות הרבה משאבים, הן לא ההעדפה הראשונה שלנו. במקום, אפשר להשתמש ב-container שמהווה וירטואליזציה למערכת ההפעלה, ללא וירטואליזציה לחומרה.[3]

קונטיינר יכול לרוץ במקביל לקונטיינרים אחרים על אותה מערכת הפעלה, ותכנית שרצה בתוכו מכילה אך ורק את התכנים שנמצאים בתוכו. באמצעותו אנו יכולים לקבל את אפליקציות שמתאימות למערכות הפעלה שונות, ללא התקנה של מכונה וירטואלית שלמה.

11.6 strace

דיברנו על כך ש-system calls הן פקודות יקרות. נרצה למצוא דרך לעקוב אחריהן במהלך ריצה של תכנית. פקודה שמאפשרת מעקב אחריהן היא strace, שמתאימה ל-linux.

[3] משתמשי mac, אתם וודאי יודעים שכדי להשתמש ב-valgrind ללא vm, צריך docker container

12 תרגול 2 - Interrupts

12.1 תאוריה מול פרקטיקה

לרוב שהתאוריה אינה מספיקה כדי להשיג ביצועים טובים

דוגמה. נביט בשתי התכניות הבאות:

```
void init_1(){
        for (h=0; h < height; ++h){
                for (w=0; w<width; ++w){
                        matrix[h][w] = 0
        }
}
void init_2 (){
        for (w=0; w<width; ++w){
                for (h=0; h < height; ++h){
                        matrix[h][w] = 0
        }
}
```

על פניו שתיהן מבצעות אותו דבר - אתחול של מטריצה, ויש להן סיבוכיות אסימפטוטית $\mathcal{O}(width \cdot height)$. יחד עם זאת, מי ירוץ יותר מהר בפועל?

נזכור שהזכרון בנוי מבלוקים רציפים, וכי המעבד שומר זכרון cache כדי לחסוך בזמן, ובכל פעם שהוא קורא זכרון הוא שומר בלוק המכיל את הבייט שביקשנו בתוך ה-Cache במחשבה שנצטרך אותו בהמשך. לכן, כאשר אנו עוברים שורה שורה, אנו חוסכים המון קריאות מהזכרון הפיסי, בעוד כאשר אנו קוראים עמודה עמודה, בכל קריאה אנחנו מבצעים קריאה פיסית לזכרון.

דוגמה. נסתכל על שני מבני נתונים deque, array ונבצע חיפוש לינארי בכל אחד מהם. מה יהיה מהיר יותר? על פניו הסיבוכיות האסימפטוטית זהה - לינארי באורך המערך. יחד עם זאת, deque שומר חוליות, זוהי מעין רשימה מקושרת ולכן בכל קריאה של איבר, תהיה קריאה של הזכרון הפיסי, בעוד קריאה של איבר במערך תהיה מה-cache שכן הזכרון רציף.

12.2 מצב משתמש ומצב גרעין - User Mode & Kernel Mode

12.2.1 תהליכים ו-I/O

הערה. המושגים הבאים שנגדיר אינם מוגדרים היטב ודורשים הקשר באזכורם.

הגדרה. תכנית היא קובץ הרצה.

הגדרה. תהליך הוא מופע של תכנית.

הגדרה. תהליך פעיל הוא תהליך שרץ כרגע על המעבד.

הגדרה. Input/Output (I/O) אוסף של תכניות או אופרציות של מכשירים המעבירים מידע מרכיבים פריפריאלים.

הגדרה. הגרעין (Kernel) הוא הליבה של מערכת ההפעלה, ויש לו שליטה מלאה על מה שקורה במערכת.

הגדרה. ה-kernel הוא תוכנה "אמינה".

הגדרה. כל תכנית שאינה ה-Kernel מוגדרת כלא אמינה.

הערה. בקורס נשתמש במושגים Kernel, OS לחילופין.

12.2.2 Modes

המעבד בעלי שני מצבים הנקבעים לפי ה-Mode Bit.

1. מצב משתמש (User Mode). מצב ללא פריבילגיות. התכנית תריץ פקודות פשוטות.
   (א) לא ניתן להריץ פקודות כמו Halt.
   (ב) לא ניתן לגשת ישירות לזכרון, אלא רק לזכרון שהוקצה על לה.

(ג) גישה מוגבלת לחומרה.

2. מצב גרעין (Kernel Mode). במצב זה מניחים שהתכונה הרצה היא "אמינה" ולכן:

(א) ניתן להריץ כל סוג של פקודה.

(ב) לגשת לכל כתובת בזכרון.

(ג) לגשת ישירות לכל רכיב חומרה.

הערה. מערכת ההפעלה קובעת מי רץ באיזה מצב. לכן יש לה שיתוף פעולה עם החומרה.

המטרה של מערכת ההפעלה היא לתת לתהליכים לרוץ. יחד עם זאת, תהליכים עלולים לדרוש שירותים שמסופקים אך ורק על ידי הגרעין, כמו למשל System Calls. על כן, על מערכת ההפעלה לעבור זמנית למצב kernel, על ידי סיפוק מנגנון System Calls. על כן זהו מנגנון עבור אפליקציות שרוצות לבקש שירות מהגרעין. דבר זה מספק ממשק בין התהליכים לבין מערכת ההפעלה. בין קריאות המערכת נמנות

read, write, close, wait, exec, fork, exit, and, kill

12.2.3 מעבר בין מצבים

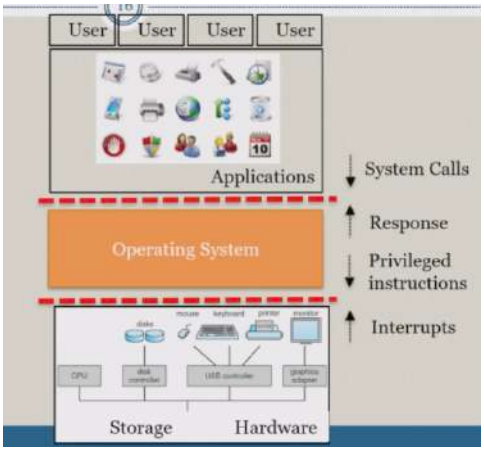
כאשר מתבצעת System Call מתבצעים השלבים הבאים:

1. מצב המעבד מוחלף ל-Kernel Mode.
2. הבקשה רצה.
3. מצב המעבד חוזר למצב משתמש.

השלב השלישי פשוט, כי זה מקטין את כמות הכוח של המשתמש. למעשה, מבין כל שלושת השלבים, השלב הבעייתי הוא השלב הראשון, שכן משתמש אינו יכול לעבור בין מצבים כרצונו, אלא רק במקרים מיוחדים. הפתרון לכך הוא Interrupts.

12.3 Interrupts

נסקור בגדול מה קורה בעת שימוש ב-Interrupt:



איור 86: אפליקציה רצה, ולפתע היא מבצעת System Call שדורשת פעולות עם פריבלגיות גבוהות. לכן מתבצעת הקריאה, וכאשר הרכיב הרלוונטי מסיים לרוץ, הוא שולח Interrupt על כך שהוא סיים ומעדכן את מערכת ההפעלה, שמעדכנת את האפליקציה.

שאלה מה החשיבות של Interrupts ביחס לשאר התוכנה במחשב?

תשובה לשם השוואה, מרבית הפונקציונליות של המחשב ממומשת ברכיבי החומרה ולא במעבד, ולכן הקשר בין מערכת ההפעלה לחומרה קריטי.

מסקנה. כיוון שכל רכיב פועל בקצב שלו, צריך לבנות מנגנון שיסנכרן את כולם, כך שהכל יהיה הרמוני. למשל, בעת קריאת מידע מקובץ, עלינו לסיים לבצע את הפעולה ורק לאחר מכן לקרוא אותו, אחרת אנו עלולים לקבל התנהגות בלתי צפויה.

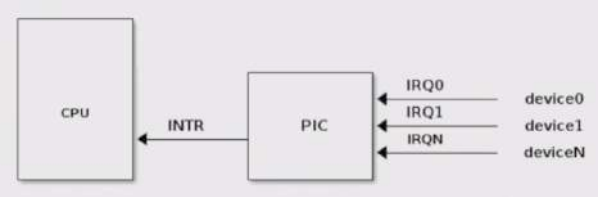
שאלה עבור בדיקה האם רכיבים סיימו את עבודתם, איזה פתרון אפשר להציע?

תשובה אפשר לעבור בצורה מסודרת על כל הרכיבים, ולשאול כל רכיב מה מצבו הנוכחי. אם הרכיב סיים, נוכל לקרוא ממנו את הבייטים הבאים.

בעיה זה אמנם יעבוד, אך זהד בזבוז של משאבי חישוב - המעבד עסוק כל הזמן בבדיקה של רכיבים במקום להריץ פקודות שימושיות. כמו כן, כשרמת העברת המידע גבוהה במיוחד, המעבד עלול לאבד מידע, עקב מבנה החומרה.

פתרון חדש Interrupts.

במקום לשאול כל פעם מי סיים, ולהטיל את האחריות על מערכת ההפעלה, נטיל את האחריות על הרכיבים. כאשר רכיב צריך את המעבד, הוא ישלח סיגנל חשמלי דרך פס בקרה מיועד שנקרא **שבב ה-Interrupt Controller**. בזאת פתרנו את בעיית הבזבוז, שכן המעבד לא עסוק בלולאה.



איור 87 : החומרה יחד עם תמיכה ב-Interrupts תראה כך, כאשר ה-PIC מסמן Programmable Interrupt Controller

עכשיו נגדיר כמה מושגים (שוב, לא פורמלי).

הגדרה. Interrupt הוא סיגנל שנשלח ל-CPU במטרה להודיע שאירוע כלשהו קרה, והוא גורם לשינוי בסדרה של ההוראות שמבוצעות על ידי ה-CPU.

הערה. Interrupt היא פקודה שאינה חלק מהקוד הרגיל שנכתב והוכן מראש. ניתן לקרוא לה APC – asynchronous procedure call, וכן, ה-Interrupt הוא אסינכרוני כיוון שהתכנית לא שולטת בו. בפועל ה-Interrupt עוצר את הפקודה הנוכחית שרצה על המעבד, ומריץ את הקוד המתאים לו.

12.3.1 סוגי Interrupts

קיימים סוגים שונים של Interrupts שמאופיינים לפי החשיבות שלהם בעיני המעבד ומי שמשתמש בהם :

1. Interrupts חיצוניים : נקראים על ידי רכיבי חומרה חיצוניים.
   (א) לחיצה על העכבר או המקלדת, הזזת העכבר, הדרייב של הדיסק שקיבל בקשה לזכרון לפני 20 דקות, השעון.
2. Interrupts פנימיים : מבוצעים במהלך ריצה של תכנית על המעבד.
3. Non Maskable Interrupts - לא ניתן לדחות או להתעלם מהן.
4. Maskable Interrupts - ניתן להתעלם או לדחות אותן.

12.3.2 טיפול ב-Interrupt

במהלך שימוש ב-Interrupt מתבצעים הדברים הבאים ומפריעים להתהליך הנוכחי שרץ :

1. קבלת ה-Interrupt.
2. שמירת המצב הנוכחי.
3. העברת השליטה ל-Kernel והרצת הקוד הרלוונטי.
4. שחזור המצב הקודם.
5. החזרת השליטה לתהליך שעצרנו.

ננתח כל אחד מהשלבים :

(1) : **קבלת ה-Interrupt** אירוע חיצוני מפריע להרצת התכנית הנוכחית, על ידי שליחת סיגנל חשמלי למעבד, שמודיע על צורך בטיפול ב-Interrupt. המעבד מטפל בסיגנל רק בסוף ה-Cycle הנוכחי, כלומר לאחר שהפקודה הנוכחית רצה, אבל לפני הפקודה הבאה נטענה מהזכרון.

(2) : **שמירת המצב הנוכחי** לפני שניתן להריץ Interrupt, צריך לשמור את המצב הנוכחי, כאילו שאנו קוראים לפרוצדורה בקוד. עלינו לשמור את ה-PC כדי שנוכל להמשיך להריץ את התהליך שעצרנו. כמו כן, הרצת הקוד המתאים ל-Interrupt ישנה רג'יסטרים של המעבד, לכן גם אותם צריך לשמור - **את כולם**.

(3) : **העברת השליטה ל-Kernel** המעבד בודק את ה-Interrupt שנשלח לאחר שינוי ביט המצב ל-Krenel Mode.
לאחר מכן, הוא ניגש ל-Interrupt Vector ומוצא את כתובת הקוד של ה-Interrupt לפי המספר הסידורי של Interrupt שמהווה אינדקס בווקטור.

(4) : **שחזור המצב הקודם** כל הרג'יסטרים ששמרנו, משוחזרים לערך שהיה להם לפני קריאת ה-Interrupt, בפרט ה-PC.

(5) : **החזרת השליטה לתהליך שעצרנו** ביט המצב מוחזר למצב משתמש או למצב שהיה לפני הקריאה ל-Interrupt.

דוגמה. (External Interrupt) נניח שהתכנית הבאה רצה:

```
proc main
        add r1, r2, r3
        sub r3, r4, r5
        and r5, r3, r6
endp main
```

במהלך ריצת התכנית המשתמש לוחץ על העכבר ולכן Interrupt נשלח. נניח שהוא לוחץ במהלך פקודת ה-sub. ה-CPU יסיים את פקודת ה-sub וישמור את כל הרג'יסטרים **וה-PC**. לאחר מכן הוא יריץ את קוד ה-Interrupt על ידי טעינת הקוד ל-PC מה-Interrupt Vector, ושינוי ביט המצב ל-Kernel. העכבר על המסך ילחץ, בכך סיימנו לטפל ב-Interrupt הותכנית יכולה להמשיך לרוץ לאחר שחזור הרגיסטרים וחזרה למצב משתמש. נעיר כי ה-Interrupt במקרה זה הוא חיצוני כי נשלח מהעכבר.

12.3.3 חריגות והפרעות פנימיות (Internel Interrupt)

הפרעות פנימיות קורות במהלך ביצוע פקודה של המעבד, תוך כדי ריצת התכנית. בשונה מהפרעות חיצוניות, שקורות בזמנים ראנדומלים.
רוב ההפרעות נגרמות כתוצאה משגיאה (חריגה) (Exceptions):

- חלוקה באפס.
- גישה לא חוקית לסגמנט בזכרון.
- הרצת פקודה ללא פריבליגיות מתאימות.
- פקודה לא חוקית.
- Page Faults עליהם נפרט בהמשך.

הפרעות אלה הן הפרעות מהתוכנה, ולא מרכיבים פריפריאלים.

כאשר הפרעה מסוג Exception מתרחשת, מערכת ההפעלה פותחת בהליך הריגת התכנית הרצה. היא שומרת את הכתובת של הפקודה שגרמה לחריגה ונותנת לתכנית צ'אנס לתקן, אם היה מי שיטפל בחריגה, התכנית מאותחלת בכתובת שנשמרה מראש, אך אם לא, היא מתה.

שאלה מה הם שלבי הטיפול בהפרעה פנימית?

תשובה אותם שלבים של הפרעה חיצונית, רק שכאן אין "קבלה" של ההפרעה, שכן היא עובדה קיימת - המעבד לא הצליח להריץ את הפקודה. במקום זאת, המעבד יוצר את ההפרעה.

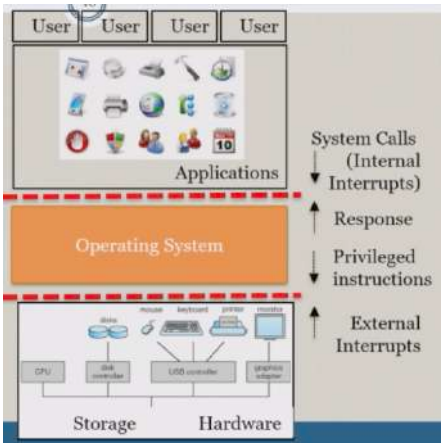
12.3.4 הפרעות פנימיות - פקודת trap

trap היא סוג של הפרעה פנימית שמופיעה כאשר תכנית רצה, רק שבשונה ממנה, היא לא מקושרת לשגיאה. הפקודה משנה את מצב הריצה למצב Kernel עבור ה-System Calls כדי שהמעבד יוכל להריץ את הפונקציות של מערכת ההפעלה.
הקוד שמטפל ב-trap רץ כדי לספק את השירות המבוקש ב-Kernel Mode, והוא אחראי לטפל בכל ה-System Calls שקיימים בעולם (Large Switch Case).
בסוף הריצה, המעבד מאתחל את התכנית בפקודה הבאה לאחר זו שגרמה ל-trap.

דוגמה. כאשר אנו שמים Breakpoint, אנו שמים פקודת int 3, שהוא interrupt בגודל כתובת של בייט אחד שמוגדר על ידי הדיבאגר להחלפה זמנית של הפקודה הרצה בתכנית, בקוד של breakpoint. מה שההפרעה עושה, הוא שכאשר התכנית תריץ int 3, מערכת ההפעלה עוצרת את התהליך.
הסינטקס לפסיקה בשפת c:

```
__asm__ ("int 3")
```

לסיכום, סוגי ההפרעות מחולקים באופן הבא:



איור 88: הרכיבים החיצוניים שולחים הפרעות חיצוניות (אסינכרוניות), בעוד האפליקציות והתוכנות הרצות שולחות הפרעות פנימיות (סינכרוניות)

דוגמה. נניח שאנו מריצים את הפקודה open ("/tmp/foo"). הקוד הוא במצב משתמש ושמור בספריה כלשהי, כיוון שהוא מעטפת ל-System Call עצמו.

הפונקציה (העוטפת) מקבלת את מספר ה-system call, הפרמטרים ושומרת אותם במקום מוגדר מראש עבור ה-kernel.

דוגמה. לאחר מכן תרוץ פקודת ה-trap (int 80) שתעבור למצב kernel וה-CPU טוען את כתובת הזכרון של קוד ה-trap, כלומר ניגש לווקטור ההפרעות במקום $128 = 0x80$. הקוד שרץ הוא מהצורה:

```
switch(sys_call_num){
        case OPEN: //...
        // handle open system call
        // ...
}
```

לאחר סיום הריצה, נשמרים ערכי ההחזרה, והשליטה למשתמש.

13 תרגול 3 - Unix Signals & Threads Implementation

13.1 Signals

סיגנלים הם הודעות שנשלחות לתהליך במטרה לידע אותו על אירוע חשוב שצריך לקרות.
סיגנלים גורמים לתהליך לעצור את מה שהוא עושה, לאחר סיום המחזור הנוכחי במעבד, ולהכריח אותו לטפל בהם באופן מיידי.
התהליך מטפל בסיגנל כרצונו, כאשר יש התנהגות דיפולטיבית שלא דורשת טיפול, מלבד כמה סיגנלים שלא ניתן לשנות את הטיפול בהם.
הערה. סיגנלים **שונים** מפסיקות.

- סיגנלים נוצרים על ידי מערכת ההפעלה ומתקבלים על ידי התהליך.
- פסיקות נוצרות ומתקבלות על ידי מערכת ההפעלה. פסיקות חומרה נוצרות על ידי החומרה, פסיקות תוכנה נוצרות על ידי התוכנה.

כמו כן, לסיגנלים ב-Unix יש שמות ומזהים.

שאלה מתי צריך סיגנלים?

דוגמאות

1. למשל, כאשר מתקבל קלט אסינכרוני מהמשתמש. כלומר, כאשר מתקבל קלט שאינו חלק מקלט שהתכנית מצפה אליו. למשל, כאשר אנו לוחצים על ctrl c בטרמינל, או כותבים kill pid.
2. המערכת או תהליך אחר. למשל, אם זמן מסוים שהתהליך הקציב עבר (SIGALARM)
3. פסיקת תוכנה (חריגה), נוצרות עקב פעולות לא חוקיות. הפסיקות מתקבלות במערכת ההפעלה וכתגובה, מערכת ההפעלה שולחת סיגנל לתהליך.
4. שליחת סיגנלים דרך המקלדת:
   (א) ctrl c - גורם למערכת לשלוח סיגנל SIGINT לתהליך שעוצר אותו לחלוטין.
   (ב) ctrl z - גורם למערכת לשלוח סיגנל SIGTSP שעוצר (משהה) את התהליך חלקית, בצורה שניתן לשחזר אותו.
   (ג) ctrl \ גורם למערכת לשלוח סיגנל SIGQUIT שהורג את התהליך אבל שומר בקובץ את המצב של הזכרון בעת ריצת התכנית.
5. שליחת סיגנלים דרך הטרמינל
   (א) kill [option] pid, אפשר להשתמש ב-kill − l כדי לראות את כל הסיגנלים.
   (ב) fg pid ממשיכה את פעולת התהליך שהופסק על ידי ctrl z, על ידי שליחה סיגנל cont.
6. זיהוי סיגנלים בתכנית ניתן לבצע באמצעות Strace.

13.1.1 שליחת סיגנל מתהליך אחד לשני

סיגנלים יכולים לשלוח הודעות מתהליך אחד לשני. כלומר מבצעים תקשורת.
דבר זה מבוצע על ידי שימוש בפונקציה

```
int kill (pid_t pid, int sig){
}
```

הסיגנלים מעבירים הודעות קבועות מראש, לא כל הודעה שנראה. יש סיגנלים ספציפיים שהוגדרו מראש עבורנו לתקשורת $SIGUSR_1, SIGUSR_2$.

13.1.2 טיפול בסיגנל

תהליך חייב לטפל בסיגנל. אם הוא לא מטפל, יש התנהגות דיפולטיבית לטיפול בו - לרוב, עצירה של התכנית, אך יש גם אפשרויות נוספות:

exit, ignore, stop, continue

לחילופין, התהליך יכול להריץ Signal Handler:

```
int sigaction(int sig, struct sigaction *new_act, strcut sigaction *old_act);
```

כאשר הסטרקט sigaction מכיל שלושה רכיבים עיקריים:

- פוינטר לפונקציה signal handler
- signal mask
- flags

סיגנלים שלא ניתן בהם יש סיגנלים שלא ניתן לטפל בהם, למשל KILL, STOP, כלומר לא ניתן לדרוס את ההתנהגות שלהם. כמו כן, אם לא הגדרנו signal handlers, הסביבה שמריצה את התכנית תשים מטפלים משלה.

כדי לשנות התנהגות דיפולטיבית אנו יכולים להשתמש בסיגנלים שהוגדרו מראש על ידי המערכת:

1. SIG_IGN - אומר לפונקציה sigaction להתעלם מהסיגנל.
2. SIG_DFL - אומר למערכת ההפעלה להחזיר את הטיפול בסיגנל לטיפול הדיפולטיבי שדרסנו עם sigaction.

Maskable Signals יתכן כי תהליך מסיים את ריצתו ומערכת ההפעלה מנקה את הזכרון, אבל באמצע אנחנו שולחים סיגנל להרוג את התהליך. לא נרצה שתהליך הניקוי גם יעצר, ולכן נרצה להשהות אתה סיגנל עד סיום הניקוי.
יכול להיות שיש תחרות בין סיגנלים, כלומר נטפל באחד בעל חשיבות גדולה יותר. לכן נרצה להיות מסוגלים לחסום אחד מהם. כיצד נעשה זאת? באמצעות f

```
int sigprocmask(int how, cons sigset_t *set, sigset_t *oldset)
```

כאשר הפרמטר הראשון הוא מה שאנו רוצים לעשות. הפמרטר השני הוא סטראקט שמייצג קבוצה של סיגנלים, הפרמטר השלישי הוא פרמטר של פלט שיחזיק את רשימת הסיגנלים שהיו חסומים עד שהפונקציה סיימה (לא בעקבותה).

How מקבל:

1. Add (SIG_BLOCK) כל מי שהיה ברשימת סיגנלי הקלט - להוסיף לרשימת הסיגנלים החסומים
2. Delete (SIG_UNBLOCK) - כל מי שהיה ברשימת הסיגנלים החסומים, שחרר אותם
3. Set (SIG_SETMASK) - לקבוע מחדש את כל רשימת הסיגנלים החסומים מאפס, כלומר לאתחל את הרשימה.

13.1.3 sigaction

מהי פונקציית הסיגנלמאסק בפנים? זו פונקציה שחוסמת את הסיגנלים שאנחנו רוצים עד לסיום ריצת הסיגנל הנדלר. אפשר לא לשלוח סיגנל מאסק ואז הסיגנל המתקבל הוא דיפולטיבי.

ברגע שקראנו ל-sigaction הוא מגדיר את ההנדלר לסיגנל להיות קבוע, עד שנשנה זאת.

נרצה להגדיר פונקציה שתהיה ההנדל לסיגנל signum. נבצע את הדבר הבא:

```
#include<stdio.h>
 #include <unistd.h>
 #include <signal.h>
void catch_int(int sig_num) {
    printf("Don't do that\n");
        fflush(stdout);
}
int main(int argc, char* argv[]) {
        // Install catch_int as the
    // signal handler for SIGINT.
         struct sigaction sa;
         sa.sa_handler = &catch_int; // set handler, rest is default
        sigaction(SIGINT, &sa, NULL); // set function to be signal handler
        // pause the process til a signal is recieved
         for ( ;; )
```

```
                    //wait until receives a signal
            pause();
        }
```

אם ננסה להריץ זאת בטרמינל ולהרוג את התהליך עם ctrl c, תודפס ההודעה "Don't Do That", כי הגדרנו הנדלר חדשים.

13.2 מימוש Threads

בהנתן מעבד והרבה משימות, כיצד נוכל ליצור תחושה של ריצה במקביל?

הדרך הסטנדרטית היא Time Sharing, שכל משימה תקבל זמן משלה.

13.2.1 User − level Threads ומימוש ספריית threads

דרך לחלק את התכנית להרצה "כאילו" במקביל, כאשר בפועל הכל רץ על אותו מעבד. אי אפשר להשיג איתם הרצה במקביל, שכן מערכת ההפעלה לא מכירה בהם.

כיוון שרק חוט אחד רץ בכל פעם, רק אחד יכול לגשת לזכרון, ולכן זו גישה בטוחה.

נוכל להחליף בין threads בפעולות הבאות:

1. לעצור את ריצת החוט
2. לשמור את מצבו הנוכחי - באמצעות sigsetjmp, שהיא כמו השמת סימניה במקום שבו אנו נמצאים.
3. לקפוץ לפעולת חוט חדש באמצעות siglongjmp שהיא כמו קפיצה למיקום של הסימניה ששמנו קודם.

בפרט, שתי הפונקציות הן למעשה זוג בלתי נפרד.

```
sigsetjmp(sigjmp_buf env, int savesigs)

// saves the stack context and cpu state i env for later use
        // pc (location in code), sp (local vars, return address)
        // not saved - global variables, dynamic allocations, values of locals
// if savesigs is not-zero, saves the current signal mask as well
//  we can later jump to this code location and state using siglongjmp
// return value - 0 if returning directly


siglongjmp(sigjmp_buf env, int val);
// jumps to the code location and resture CPU state specified by enc
// the jump will take us into the location in the code where the sigsetjmp has been
   called
// if the signal mask was saved in sigsetjmp, it will be restored as well
// the return value of sigsetjmp after arriving fom siglongjmp will be the user-defined
   val
```

דוגמה. נביט בשני חוטים, שלכל אחד משימה משלו: כאשר $\text{env}[0], \text{env}[1]$ מאותחלים לשם ריצת התכנית ושימוש בחוטים

```
//Thread 0:
void switchThreads() {
   static int curThread = 0;
   int ret_val =     sigsetjmp(env[curThread],1);
   if (ret_val == 5) {
       return;
   }
   curThread =      1 - curThread;
   siglongjmp(env[curThread],5);
}

//Thread 1:
```

```
void switchThreads() {
    static int curThread = 0;
    int ret_val =     sigsetjmp(env[curThread],1);
    if (ret_val == 5) {
        return;
    }
    curThread =       1 - curThread;
    siglongjmp(env[curThread],5);
}
```

מה קורה פה בדיוק? בהתחלה החוט הראשון רץ, ובקריאה ל-sigsetjmp נשמור את המצב שלו ונחזיר 0, ולכן לא נכנסים לתנאי, אם כך $\text{curThread} = 1$ ולכן כשנעשה siglongjmp עכשיו, נקפוץ לחוט מספר 1. עתה נתחיל בריצת התכנית שוב עם חוט 1, רק שהמשתנה הסטטי לא יאותחל פעם נוספת. לכן ב-set נחזיר 0, כאשר אנו שומרים את המצב של חוט 1, ושומרים את הסיגנלים עם ארגומנט 1. לכן שוב לא ניכנס לתנאי. אם כך $\text{curThread} = 0$, ונקפוץ שוב להקשר הריצה של חוט 0 ששמרנו קודם. אבל הפעם, בגלל שקפצנו מ-siglongjmp ערך ההחזרה הוא הארגומנט שהיא קיבלה, היינו 5, ולכן ניכנס לתנאי ונעצור.

14 תרגול 4 - עבודה עם חוטים (Cuncurrency)

14.1 ניהול חוטים באמצעות pthreads.h

pthreads.h זוהי ספריה ל-user level threads שניתן ליצור באמצעותה threads.

לכל תהליך יש main thread שרץ, והוא יכול ליצור חוטים נוספים:

```
int pthread_create
(
pthread_t *thread,
const pthread_attr_t *attr=NULL,
void *(*start_routine) (void*),
void *arg
);
```

ניתן לבצע terminate באמצעות pthread_exit (void * status).

דוגמה. נראה תכנית של כמה חוטים שמריצים אותו קטע קוד:

```
#define NUM_THREADS 5

void *PrintHello(void *arg){
        int *index = arg;
        printf("\nThread %d: Hello World!\n", *index);
        pthread_exit(NULL);
}

int main(int argc, char *argv[]){
        pthread_t threads[NUM_THREADS];
        int res, t;
        for (t = 0; t < NUM_THREADS; t++){
                printf("Creating Thread %d\n", t);
                res = pthread_create(&threads[t], NULL, PrintHello, (void*)&t);
                if (res < 0 ){
                        printf("Error\n");
                        exit(-1);
                }
        }
        return 0;
}
```

האם הקוד יעבוד? דווקא לא! אף אחד לא הבטיח שה-threads יסיימו לפני ה-main − thread. בשביל זה יש את הפונקציה

pthread_join (pthread_t thread, void **value_ptr)

אשר מחכה ל-thread שיסיים את ריצתו, כלומר חוסמת את ה-חוט שקרא לה עד שתסיים, במקרה שלנו ה-main − thread. במידה והוא סיים, היא חוזרת מיד. לכן, התכנית תהפוך ל-

```
#define NUM_THREADS 5

void *PrintHello(void *arg){
        int *index = arg;
        printf("\nThread %d: Hello World!\n", *index);
        pthread_exit(NULL);
}

int main(int argc, char *argv[]){
```

```
        pthread_t threads[NUM_THREADS];
        int res, t;
        for (t = 0; t < NUM_THREADS; t++){
                printf("Creating Thread %d\n", t);
                res = pthread_create(&threads[t], NULL, PrintHello, (void*)&t);
                if (res < 0 ){
                        printf("Error\n");
                        exit(-1);
                }
        }
        // main thread waits for the other threads
        for(t=0; t<NUM_THREADS; t++) {
           res = pthread_join(threads[t], (void **) &status);
       if (res < 0) {
         printf("ERROR\n");
         exit(-1);
          }
          printf("Completed join with thread %d status= %d\n", t, *status);
        }
        return 0;
}
```

14.2 Mutex

סנכרון זה לא דבר קל. למשל, קטע הקוד הבא שירוץ במקביל לשני חוטים, הוא מקור להרבה בעיות:

```
//thread 1
int a = counter;
a++;
counter = a;
// thread 2
int b = counter;
b--;
counter = b;
```

אם הריצה היא במקביל יתכן שהערך החדש של המונה הוא $\text{counter}-1, \text{counter}, \text{counter}+1$, שזה בוודאי לא מה שרצינו. לכן עלינו לספק מנגנון להגנה על קטע הקוד.

14.2.1 בעיית קטע הקוד הקריטי

- נניח שיש לנו n תהליכים $P_0, P_1, \ldots, P_{n-1}$.
- אין הנחות על מהירות המעבדים.
- אין הנחות על ביצוע התהליכים.

בעיה. מימוש מנגנון כללי שיאפשר גישה לקטע הקוד הקריטי.

בהנתן אלגוריתם שפותר את הבעיה, נמודד כמה הוא טוב לפי הקריטריונים שראינו בהרצאה:

1. Mutual Exclusion
2. Progress
3. Starvation Freedom
4. Generality
5. No Blocking In The Reminder

דוגמה. (האלגוריתם של Perterson)

```
bool want[0] = false;
bool want[1] = false;
int turn;
// thread 0
i = 0;
want[i] = true;
turn = 1-i;
while (want[1-i] && turn == 1-i){
        want[i] = false;
}

// thread 1
i = 1;
want[i] = true;
turn = 1-i;
while (want[1-i] && turn == 1-i){
        want[i] = false;
}
```

מה קורה כאן? למעשה, החוט הנוכחי שרץ מסמן שהוא רוצה לגשת לקטע הקוד, ומניח שהחוט האחר גם רוצה. הוא מחכה שהחוט השני כבר לא ירצה לגשת, או שהחוט יסמן לו שהתור שלו. האלגוריתם מקיים את חמשת התנאים.

אם במקום $1-i$ נחליף את השורה ב-i, האם זה יעבוד? (ספויילר, לא).

14.2.2 אלגוריתם למימוש Mutex - Mutual Exclusion

השימוש הנפוץ בנעילה למשאב משותף הוא ב-Mutex. באופן כללי, שימוש באובייקט Mutex נעשה כדלקמן:

1. יצרית ואתחול של אובייקט.
2. חוטים שונים מנסים לנעול את האובייקט.
3. רק אחד מצליח, והוא האחראי עליו.
4. החוט האחראי מבצע את קטע הקוד שרצה.
5. החוט האחראי משחרר את האובייקט.
6. חוט אחר לוקח אחריות על המנעול וחוזר על התהליך.

יצירת ושחרור של Mutex מחריזים על Mutex או סטאטית או דניאמית:

```
// statically
pthread_mutex_t my_mutex = PTHREAD_MUTEX_INITILIZER;
// dinamically
pthread_mutex_init(mutex, attr);
// allows setting mutex attributes
```

השחרור נעשה באופן הבא:

```
pthread_mutex_destroy(mutex)
```

נעילה של Mutex נעילה מתבצעת באופן הבא:

```
pthread_mutex_lock(*mutex)
```

כאשר אם הוא כבר ננעל, הפונקציה תעשה block לחוט עד שהמנעול ישוחרר.

שחרור של Mutex היא מחזירה שגיאה כאשר ה-mutex לא ננעל.

```
pthread_mutex_unlock(*mutex)
```

דוגמה. על קטע הקוד שראינו קודם, ניתן להגן באופן הבא:

```
// thread 1
lock(a_mutex);
lock(b_mutex);
a=b+a;
b=b*a;
unlock(b_mutex);
unlock(a_mutex);

// thread 2
lock(b_mutex);
lock(a_mutex);
b=a+b;
a=b*a;
unlock(a_mutex);
unlock(b_mutex);
```

פתרנו את בעיית ה-Mutual Exclusion. אבל מה קרה כאן? Thread 1 נעל את a ו-Thread 2 נעל את b, שניהם מחכים לשחרור של השני, ולכן יש כאן DeadLock.

14.3 Monitors

mutex תוכנן בדיוק בשביל ה-Mutual Exclusion. אבל לרוב, יש סנכרון יותר רחב שאנו רוצים. למשל, נרצה שבנקודה מסויימת בקוד כל החוטים יישרו קו (Barrier), Reader − Writers.

דוגמה. נתון משתנה משותף var שמי שאחראי על האתחול שלו זה חוט 2. אנו רוצים שרק לאחר שחוט 2 ירוץ, ירוץ חוט 1. כיצד נעשה את זה? למשל, באמצעות Mutex

```
// thread 1
while (true) {
  pthread_mutex_lock(     &mut_var);
  if (canUseVar) {
    // use var     canUseVar = false;
  }
  pthread_mutex_unlock(   &mut_var);
}
// thread 2
pthread_mutex_lock(     &mut_var);
init(var);
canUseVar = true;
pthread_mutex_unlock(   &mut_var);
```

יחד עם זאת, השימוש כאן לא יעיל, שכן Thread 1 ממשיך לרוץ בלולאה אינסופית עד שהתנאי מתקיים.

כדי להימנע מהמקרה הנ"ל, משתמשים ב-Monitors, שמונעים את ה-Busy Waiting. נעשה זאת באופן הבא:

```
// thread 1
pthread_mutex_lock(     &mut_var);
if (canUseVar) {
  pthread_cond_wait(      &cv_var, &mut_var);
}
// use var canUseVar = false;
pthread_mutex_unlock(   &mut_var);
```

```
// thread 2
pthread_mutex_lock(      &mut_var);
init(var);
canUseVar = true;
pthread_cond_broadcast( &cv_var);
pthread_mutex_unlock(    &mut_var);
```

כאן החוט הראשון לא רץ ומחכה, אלא ישן, וכאשר חוט 2 סיים, הוא שולח סיגנל שמעיר את חוט 1, וככה נחסכת הלולאה האינסופית.

pthread מכילה פונקציות ל-monitoring:

- $\text{pthread\_cond\_wait}(cv, mutex)$, נקראת על ידי חוט כאשר הוא רוצה להיכנס למצב blocked ולחכות עד שהתנאי יהפוך ל-ture. מניחים שהוא נעל את ה-mutex, ושהוא באמת צריך לחכות.
- wait, גורמת לחוט לשחרר את המנעול, ועושה block עד שמעירים אותו על ידי $\text{pthread\_cond\_signal}(cv)$, מחוט אחר. כאשר הוא מתעורר, הוא מחכה עד שהוא יכול לקחת בעלות על המנעול, וכאשר זה קורה, הוא חוזר מ-$\text{pthread\_cond\_wait}(cv, mutex)$.
- $\text{ptheread\_cond\_broadcast}(cv)$ מעירה את כל החוטים שמחכים לתנאי.

14.4 Barrier

נביט בדוגמת הקוד הבאה:

```
#define NTHREADS 5
pthread_mutex_t *n_done_mutex;
pthread_cond_t *barriar_cv;
int n_done = 0;
int main() {
// Checking of return values omitted for brevity
   pthread_t tids[NTHREADS];
   int i;
   void *retval;
   pthread_mutex_init(&n_done_mutex, NULL);
   pthread_cond_init(&barriar_cv, NULL);
   for (i = 0; i < NTHREADS; i++){
       pthread_create(&tids[i], NULL, barrier, (void *) &i);
        }
   for (i = 0; i < NTHREADS; i++) {
       pthread_join(tids[i], &retval);
        }
   printf("done\n");
}

void *barrier(void *arg) {
   // Checking of return values omitted for brevity
   int* id = (int *) arg;
   printf("Thread %d -- waiting for barrier\n", id);
   pthread_mutex_lock(n_done_mutex);
   ndone = ndone + 1;
   if (ndone < NTHREADS) {
       pthread_cond_wait(barriar_cv, n_done_mutex);
   }    else {
       pthread_cond_broadcast(barriar_cv);
   }
   pthread_mutex_unlock(n_done_mutex);
   printf("Thread %d -- after barrier\n", id);
}
```

הפונקציה barrier חוסמת את כל החוטים עד שהאחרון משחרר אותם.

14.5 Semaphores

זוהי הרחבה ל-Mutex, כלומר מאפשר נעילה עבור כמה חוטים.

מאתחלים סמפור באמצעות כל ערך, שמייצג כמה חוטים יכולים לגשת במקביל. חוט נחסם אך ורק עם הערך הוא אפס.

פונקציות שימושיות לאתחול:

```
sem_t semaphore;
int sem_init(sem_t *sem, int pshared, unsigned int value);
int sem_destroy(sem_t *sem);
```

כיצד מעדכנים סמפור?

```
std::atomic<int> counter;
void *foo(void *arg){
        for (int i = 0; i < 1000; ++i){
                counter += 1;
        }
}
int main() {
        pthread_t threads[5];
   for (i = 0; i < 5; i++){
       pthread_create(&threads[i], NULL, foo, NULL);
        }
   for (i = 0; i < NTHREADS; i++) {
       pthread_join(threads[i], NULL);
        }
   printf("counter: %d\n", counter.load());
        return 0;
}
```

כאשר השימוש ב-counter std :: atomic < int > מאפשר העלאה מבלי לחסום ישירות את קטע הקוד הקריטי.

15 תרגול 5 - מקביליות (Cont.) Concurrency

15.1 בעיית ה-Bounded Buffer

יש לנו יצרן (Producer) וצרכן (Consumer). ה-Buffer משותף וציקלי ויש בו N מקומות. כיצד נממש קריאה וכתיבה אליו?
נשתמש בשלושה Semaphores:

- Mutex - מאותחל ל-1.
- FillCount - מאותחל ל-0.
- EmptyCount - מאותחל ל-N.

הבא נכתוב קוד:

```
// producer
while (true) {
        // pruduce an iterm
        down(emptyCount);
                down(mutex);
                        // add the item to buffer
                up(mutex)
        up(fillCount)
}

// consumer
while(true) {
        down(fillCount);
                down(mutex);
                        // remove an item from buffer
                up(mutex);
        up(emptyCount);
        // consume an item
}
```

15.2 בעיית ה-Dining – Philosophers

דיקסטרה (כן אותו דיסקטרה מ-APSP) כתב את בעיית הפילוסופים כתשובה לשאלה במבחן.
טוני הור פירמל את הבעיה למה שהיא מוכרת כיום.
נפרמל את הבעיה:

מידע משותף קערת אורז ממנה אוכלים. Semaphores c [5] שמאותחלים לערך 1.

נציע את הקוד הבא:

```
While (True){
        down (c[i])
        down (c[(i+1 % 6]);
        eat
        up(c[(i+1)%6]);
        up(c[i]);
        think
}
```

האם זה פתרון טוב? לא. מדוע? יתכן שיש DeadLock. למשל כאשר מחכים $0 \to 5 \to 4 \to 3 \to 2 \to 1 \to 0$.
הסיבה היא שהפתרון הוא סימטרי, כלומר כל הפילסופים מריצים את אותו הקוד, ולכן אנו מכניסים את עצמו למלכודת.
כיצד נשבור את הסימטריה? נשבור את הסימטריה באמצעות אלגוריתם הסתברותי!

15.2.1 אלגוריתם Lehmann – Rabin

```
reapat:
if coinflip() == 0 then //  randomly decide on a first chopstic
        first = left
else
        first = right
end if
wait until c[first] == false //  wait until it is available
c[first] = true
if c[~first] == false then // if second chopstic is available take it
        c[~first]  = true
        break;
else
        c[first] = false
end if
end repeat
Eat
c[left] = false;
c[right] = false;
```

בתוחלת אין DeadLock.

15.3 בעיית ה-Readers – Writers

נניח שיש מבנה נתונים משותף לכמה תהליכים הרצים במקביל. יש לנו שני סוגי תהליכים:

1. Readers : רק קוראים מידעף לא משנים את המבנה.
2. Writers : יכולים גם לקרוא וגם לכתוב למבנה.

בעיה. סנכרון התהליכים כך שקוראים לא יגשו במקביל לקובץ כאשר כותבים ניגשים אליו ולהפך.

כיצד נפתור את בעיה?

נציע את הפתרון הבא. נחזיק Semaphores:

- ReadCount שמאותחל ל-0 - מספר הקוראים.
- ReadCount_mutex שמאותחל ל-1 - מגן על ReadCount.
- write שמאותחל ל-1 - מוודא שהכותבים לא ניגשם למידע באותו זמן שהקוראים ניגשים.

מכאן נקבל את קטע הקוד הבא:

```
// writer thread
while (true){
        down(write);
                // writing is performed
        up(write);
}

// reader thread
while(true){
        down(readCount_mutex);
        readCount++;
        if (readCount == 1){
                down(write); // lock from writers
        }
        up(readCount_mutex);
        // readind is performed
```

```
        down(readCount_mutex);
        readCount --;
        if (readCount == 0){
                up(write);
        }
}
```

מעבר על הקוד מראה שהוא אכן עובד.

בעיה. אין לנו עדיפות לכותבים.

נוסיף סמפורס:

- Read שמאותחל ל-1 - מונע מקוראים להיכנס למבנה כאשר כותב רוצה לכתוב.
- WriteCount שמאותחל ל-0 - מונה כותבים שמחכים.
- writeCount_mutex שמאותחל ל-1 - שולט בעדכון של WriteCount.
- Queue - מאותחל ל-1 - רק הקוראים משתמשים בו.

מכאן נקבל את קטע הקוד הבא:

```
// writer thread
while(true){
        down(writeCount_mutex);
        writeCount++; // countes number of waiting writers
        if (writeCount == 1){
                down(read); // lock from readers
        }
        up(writeCount_mutex);
        down(write);
        // writing is performed
        up(write);

        down(writeCount_mutex);
        writeCount--;
        if (writeCount == 0){
                up(read);
        }
        up(writeCount_mutex);
}

// reader thread
// we don't want to allow more than one reader at a time
while(true){
        down(queue); //  only one reader  gets the queue and thus the writer is 50
            percents likely to control read
                down(read);
                        down(readCount_mutex);
                                readCount++;
                                if (readCount == 1){
                                        down(write);
                                }
                        up(readCount_mutex);
                up(read);
        up(queue);

        // reading is performed

```

```
        down(readCount_mutex);
        readCount--;
        if (readCount == 0){
                up(write);
        }
        up(readCount_mutex);
}
```

16 תרגול 6 - תזמון Scheduling

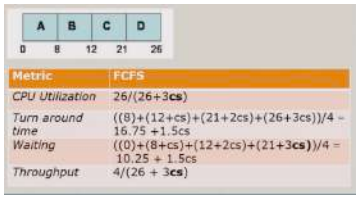
יש כמה קריטריונים בתזמון:

- נרצה למקסם את:
  - ניצול המעבד (Utilization). שיהיה כמה שיותר עסוק.
  - (Throughput). מספר התהליכים שמסתיימים בכל יחידת זמן.
- נרצה למזער את:
  - (Waiting Time). זמן ההמתנה של תהליך ב-Ready Queue
  - (Turnaroung Time). הזמן לביצוע תהליך.

16.1 אלגוריתם ה-FCFS

ניתן לתהליך שהגיע זה עתה לרוץ. כלומר נריץ על פי סדר ההגעה.

דוגמה. נביט בדוגמא הבאה תחת אלגוריתם ה-FCFS שראינו בהרצאה:



איור 89: חישוב הפרמטרים עבור ארבעה תהליכים.

הבעיה שעבודות ארוכות וקצרות יביאו לזמן המתנה ממוצע ארוך. כמו כן, עלינו לזכור שהאלגוריתם לא עוצר עבודות באמצע, אלא אם מערכת ההפעלה החליטה לעצור אותו.

16.2 אלגוריתם ה-SJF

נבחר את מי להריץ על פי זמן הריצה הקצר ביותר. קצרות בהתחלה, ארוכות בסוף. אם יש משימות בעלות אותו זמן ריצה, נשבור את השוויון על פי FCFS.

אם אנחנו יודעים את כל המשימות מראש, הוא אופטימלי (הוכחנו בהרצאה). יש לנו זמן המתנה מינימלי ביחס לכל שאר אלגוריתמי ה-$Off-line$.

בעיה. זמן ביצוע העבודה לא בהכרח ידוע מראש. כמו כן, אין כאן הוגנות ותתכן Starvation, שכן מתעדפים עבודות קצרות יותר.

אלגוריתם זה הוא Preemptive, כלומר, אם עבודה חדשה שהגיעה בעלת זמן ריצה קטן יותר מהעבודה הנוכחית, נעצור את העבודה הנוכחית ונעבור לחדשה! לכן יש יותר Overhead בגלל שיש Context Switch.

16.3 אלגוריתם ה-Priority

המעבד מוקצה לתהליך עם העדיפות הגדולה ביותר. עדיפות נתנת לכל תהליך כמספר, כאשר ערך נמוך מציין עדיפות גבוהה.

עדיפויות זהות, מוקצות על פי FCFS. למעשה SJF הוא מקרה פרטי, שכן הוא מתעדף תהליכים עם זמן ריצה נמוך וזו העדיפות.

שאלה כיצד קובעים עדיפויות?

תשובה למשל, פנימית. מערכת ההפעלה תתן עדיפות ותשמור אותה בתוך ה-PCB. אבל אפשר גם חיצונית, למשל המשתמש ייתן עדיפות ידנית לכל תהליך.

אלגוריתם זה יכול להיות Preemptive ויכול גם לא להיות.

בעיה. יש הרעבה, שכן תהליכים בעלי עדיפות גבוהה עלולים לא לרוץ לעולם.

פתרון. Aging - נשנה את העדיפות לפי זמן ההמתנה.

16.4 אלגוריתם Round – Robin (RR)

מגדירים מראש גודל קוואנטום. תור ה-Ready הוא FIFO, והמעבד סורק אותו.

כאשר תהליך מקבל את המעבד, הוא מקבל אותו לכל היותר לגודל לקוואנטום אחד. למעשה, יש שני מקרים בהם מחליפים תהליכים:

1. התהליך שחרר את המעבד בעצמו, למשל, עקב סיום ריצתו, שימוש ב-I/O.
2. עבר קוואנטום אחד בזמן הריצה ולכן צריך לבצע Context Switch.

החסרון הוא ה-Overhead בגלל ה-Context Switchs הרבים שמתבצעים.

היתרון הוא שאין Starvation.

עוד חסרון הוא Completion Time גבוה. למשל כאשר כל התהליכים בעלי אותו זמן ריצה, הם יחכו שכולם יסיימו, בעוד אלגוריתמים אחרים יתזמנו אותם אחרת.

בהשוואה ל-FCFS מקבלים שהוא בין המקרה הגרוע למקרה הטוב. למשל עם העבודות מגיעות בסדר לפי זמן הריצה שלהן באופן עולה, הוא אופטימלי. אבל אם הסדר יורד, הוא הכי גרוע שיכול להיות, ולכן RR נותן תוצאת ביניים טובה.

16.5 אלגוריתם ה-Multilevel – Queue

מחלקים את ה-Ready Queue לתתי תורים שלכל אחד אלגוריתם תזמון משלו. כלומר כל חלק בתור מסמל עדיפות מסויימת.

האלגוריתם מאפשר Preemption הגורם ל-Starvation. הפתרון הוא להשתמש ב-Time Slices כלומר לתת לכל תור אחוז שימוש.

למשל, נחלק את המשימות לשתי קבוצות:

1. Foreground/Interactive - אנחנו מריצים בעצמנו.
2. Background/Batch - משימות שרצות ברקע.

אבל אפשר לעשות גם חלוקה ל-5 תורים ולכל מספר כרצוננו.

תהליך יכול לזוז בין תורים שונים. למשל, תור מחכה הרבה זמן, מקבל עדיפות גבוהה יותר.

16.6 Supercomputers

מחשבים קטנים עם מעבדים בעלי מערכת הפעלה משלהם (לתמיכה ב-System Calls) שאחראים לביצוע משימות מסויימות, אותן מתזמנת מערכת ההפעלה. הם רצים במקביל כדי לאפשר למשתמשים חיצוניים לבקש להריץ עליהם משימות. על כל התזמון שולטת מערכת ההפעלה הגלובלית.

AWS שירות של אמזון למחשבי על חזקים במיוחד שאפשר להשתמש בהם דרך האינטרנט - **אלה שירותי ענן**. למשל, כשאנו רואים סדרה בנפליקס, כנראה משתמשים מאחורי הקלעים ב-AWS.

נרצה אם כך לבנות מתזמן למחשבי-על.

כשבונים אלגוריתם כזה משתמשים ב-Backfilling: כלומר, כאשר אין לנו מספיק משאבים לתהליך מסויים, אבל מספיק בשביל תהליך חדש יותר, נאפשר להתליך החדש לרוץ וכשיהיו משאבים לתהליך החדש, ניתן לו לרוץ.

16.6.1 אלגוריתם ה-Easy

נריץ את התהליכים לפי FCFS.

במידה שאין למשימה הנוכחית מספיק מקום, נקצה לה מקום לעתיד. במידה שיש חורים, נשתמש בהם בשביל משימות קטנות (Backfill).

כמו כן, כאשר משתמשים רוצים להריץ תהליך, הם צריכים לספק הארכה לזמן הריצה שלו ואת מספר התהליכים הדרושים להרצה שלו.

17 תרגול 7 - ניהול זכרון Memory Management

17.1 מספרים בסיסיים

- $\text{Byte} = 8\text{Bit}$
- $\text{KB} = 2^{10}\text{Bytes}$
- $\text{MB} = 2^{20}\text{Bytes} = 2^{10}\text{KB}$
- $\text{GB} = 2^{30}\text{Bytes}$.

דוגמה. $\frac{4\text{GB}}{8\text{KB}} = \frac{2^{32}\text{Bytes}}{2^{13}\text{Bytes}} = 2^{19}$.

כמה מספרים בינארים ניתן לייצג באמצעות מספר בינארי בעל d ספרות? 2^d.

מה הערך הדצימלי של 1101? נחשב $1 \cdot 2^0 + 0 \cdot 2^1 + 1 \cdot 2^2 + 1 \cdot 2^3 = 13$.

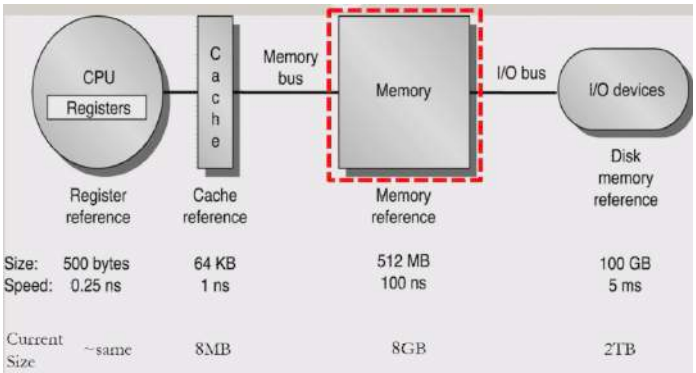
כמה זה 12 בבינארי? נבצע מעבר בסיסים:

$$12 \mod 2 = 0$$
$$\left\lfloor \frac{12}{2} \right\rfloor = 6$$
$$6 \mod 2 = 0$$
$$\left\lfloor \frac{6}{2} \right\rfloor = 3$$
$$3 \mod 2 = 1$$
$$\left\lfloor \frac{3}{2} \right\rfloor = 1$$
$$1 \mod 2 = 1$$
$$\left\lfloor \frac{1}{2} \right\rfloor = 0$$

קיבלנו אפוא את המספר 1100.

17.2 היררכיית הזכרון

אנו נדבר על הזכרון הראשי, ולא נרחיב על ה-Cache.



איור 90: היררכיית הזכרון

17.3 כתובת וירטואלית

הכתובות שאנו רואים בתכנית c כלומר ערכים שפוינטרים מאחסנים, אלה כתובות וירטואליות אותן המעבד ממפה לכתובות פיסיות.

17.4 ארכיטקטורת סגמנטים

אנו מחלקים את התכנית למספר סגמנטים: $\text{Code, Heap, Stack, Data} - \text{Structures}$ ושומרים טבלה של סגמנטים, כדי לגשת לכל סגמנט. ככה אפשר לשמור את התהליך על ידי חלוקה שלו למקומות שונים בזכרון ולא כרצף.

כתובת לוגית מורכבת משני חלקית $\langle \text{Segment Number} \mid \text{Offset} \rangle$.

אנו נשמור טבלת סגמנטים שבאינדקס ה-Segment Number תכיל את ה-Base ואת ה-Limit ככה נקבל כתובת פיסית $\langle \text{Base} \mid \text{Offset} \rangle$ ונוכל לבצע ואלידציה באמצעות ה-Limit. כלומר, כאשר ניגש לכניסה המתאימה בטבלה, נשווה את ה-Offset ל-Limit ואם קיבלנו שהוא גולש החוצה, נזרוק חריגה.

כדי לגשת אליה, המעבד שומר רג'יסטר יעודי STBR שמצביע למיקום הטבלה. כמו כן, הוא ישמור רג'יסטר עם אורך טבלת הסגמנטים, שהיא גם כל הסמגנטים של התהליך.

בפועל, אנחנו צריכים יותר ביטים עבור כל סגמנט בטבלה:

- Validation Bit - האם מדובר בסגמנט חוקי, כלומר סגמנט ששמור בזכרון התכנית. אם הוא 0, צריך לזרוק Exception ולטעון אותו.
- הרשאות Read/Write/Execute.

בעיה. אנו עלולים לקבל פרגמנטציה, כלומר רווח של זכרון ריק בין חלקים שונים של תהליכים, שימנע שימוש שלו עבור חלקים גדולים יותר.

17.5 Paging

נחלק את מרחב הכתובות לבלוקים בגודל קבוע שנכנה Frames. הגודל יהיה חזקה של 2.

כל תהליך יחלוק לבלוקים בגודל של Frame, שנכנה Pages. נשמור עבורו טבלה של Pages והכתובת הלוגית שלו תהיה

$$\langle \text{PageNumber} \mid \text{Page Offset} \rangle$$

אם מרחב כל הכתובות בגודל 2^m וגודל ה-Page הוא 2^n נקבל ש-

$$|\text{Page Offset}| = n\text{bits}, |\text{PageNumber}| = m - n \text{ bits}$$

במקרה זה אין פרגמנטציה חיצונית, אלא רק פרגמנטציה פנימית.

היות שהמרחב מחולק לבלוקים בגודל קבוע, שהם יחסית קטנים, אנו מאבדים את היכולת לקרוא להם Stack/Heap. כל סגמנט יורכב מכמות Pages שונה.

17.5.1 Page Table

נשמור ביטים נוספים:

- Valid Bit - האם הדף ממופה ל-Frame.
- Modified (Dirty Bit) - האם הדף השתנה.
- Used Bit - מתי הייתה הפעם האחרונה שניגשו אליו.
- $\text{Access Permissions} - \text{R, W, X}$ - הרשאות גישה.

כאשר ה-$\text{Valid Bit} = 0$ צריך להביא את הדף מהדיסק לזכרון. במקרה זה תזרק חריגת Page Fault, ומערכת ההפעלה תטען את הדף לזכרון, **ולא תהרוג את התהליך**.

כשנרצה לטעון אותו, נשתמש באלגוריתם החלפה.

נבחין שככל שמספר התהליכים גדל, כך גם כמות הזכרון שמוקצית לטבלות הדפים. לכן עלינו למצוא דרך לייעל את זה.

17.5.2 Hierarchial Page Table

אנו נרצה לבצע חלוקה כללית של המרחב שתהיה קבועה, וממנה לקבל חלוקה פרטית עבור התהליך. לכן נבנה עץ של טבלות.

למשל, עבור עץ עם שתי רמות, נשמור טבלה כללית של טבלות. ככה, הטבלה הכללית תשמר בזכרון, ושאר הטבלות, יהיו בשימוש רק במידת הצורך, ולכן חלק מהזמן **יהיו שמורות בדיסק** ולא בזכרון הראשי, מה שיחסוך הרבה זכרון[4].

אנו נחלק את הכתובת הלוגית לכמה חלקים, כך שתתבצע גישה לטבלה, ממנה לעוד טבלה וממנה לדף הרצוי. כלומר נקבל כתובת

$$\langle p_1 \mid p_2 \mid d \rangle$$

ניגש לטבלה הראשונה במקום ה-p_1 ממנה נקבל כתובת שניגש אליה עם היסט p_2 וממנה ניקח את הדף, שבו נהיה בהיסט d.

החסרון בשיטה זו הוא שנדרשות 3 גישות לזכרון כדי להגיע לדף.

חלוקת הביטים תהיה 22 ביטים ל-$p_1 + p_2$ ו-10 ביטים עבור ה-offset.

[4] אנחנו מזמינים אתכם לקרוא על מבנה הנתונים Van Emde Boas Tree שמממש רעיון דומה, ויכול לתרום להבנה.

17.5.3 Inverted Page Table

נשמור טבלה בגודל הזכרון הפיסי, שתהיה משותפת לכל התהליכים, שתבצע מיפוי הפוך. לכל מסגרת, היא תמפה את ה-Page המתאים.

עבור כתובת וירטואלית $\langle p \mid d \rangle$ נוסיף נוסיף שדה $\text{pid} = \text{process id}$ ונקבל כתובת $\langle \text{pid} \mid p \mid d \rangle$, כדי לקבל את ה-frame, נמצא את האינדקס של הזוג $\langle \text{pid} \mid p \rangle$ בטבלת הדפים. כלומר, נעבור עליה מההתחלה עד הסוף עד שנגיע לזוג. נקבל אינדקס i שייצג את מספר ה-frame. הכתובת הפיסית תהיה $\langle i \mid d \rangle$.

הבעיה היא שהחיפוש לא יעיל במיוחד, הן בגלל שהוא לינארי בגודל הטבלה, והן בגלל שהוא דורש גישות לזכרון.

17.6 Translation Lookaside Buffer (TLB)

כדי להתמודד עם הגישות הרבות לזכרון כדי לחפש את ה-frame עבור Page מסוים, יש זכרון cache במעבד שמכיל מבנה נתונים שאנו קוראים לו ה-TLB. מבנה נתונים זה הוא 64 כניסות של frames.

ב-TLB אנו שומרים גם את מספר התהליך שהוא שייך לו. במידה שלא נמצא הדף ב-TLB מנסים לגשת לטבלת הדפים בזכרון הפיסי, אם הוא גם לא נמצא שם, יש Page Fault והדף נטען מהדיסק לזכרון הפיסי, או ל-TLB.

במקרה של עץ טבלות בעומק 2, אנו נחפש ב-TLB את $p_1 \mid\mid p_2$, וכך נחסוך שתי גישות לזכרון.

18 תרגול 8 - אלגוריתמי החלפת דפים (Pages Replacement Algorithms)

בבניית אלגוריתם להחלפת דפים, אנו רוצים למזער את ה-Page Miss Rate כלומר הזמן היחסי שאנו מחפשים Page והוא לא נמצא ב-Ram.

במקרה בו כל המסגרות תפוסות, אפשר לבחור Page תפוס ולבצע החלפה, לפעולה זו אנו קוראים Evict, ולדף הנבחר אנו קוראים Victim. האלגוריתמים שראינו עד כה:

- אופטימלי.
- FIFO.
- אלגוריתם השעון, (Second Chance FIFO).
- NRO (not recently used).
- LRU (Least Recently Used).
- $\text{Pseudo} - \text{LRU}$.
- LFU (Least Frequnelty Used).
- Random.

18.1 אלגוריתם אופטימלי (Belady's Algorithm)

נבחר את הדף שנשתמש בו הכי מעט זמן בעתיד, מה שנותן את השגיאה הקטנה ביותר.

דוגמה. נסתכל על הדפים שצריכים בסדר הבא:

$$0, 2, 1, 6, 4, 0, 1, 0, 3, 1, 2, 1$$

ונניח כי יש לנו 4 מסגרות. תחילה נקבל זכרון כך:

$$\boxed{\begin{matrix}0\\2\\1\\6\end{matrix}} \xrightarrow{6\to4} \boxed{\begin{matrix}0\\2\\1\\4\end{matrix}} \xrightarrow{0\to3} \boxed{\begin{matrix}3\\2\\1\\4\end{matrix}}$$

כמות הפעמים שהיה Page Fault היא 6 ולכן השגיאה היא $\frac{|\text{Page Faults Amount}|}{|\text{Pages Amount}|} = \frac{6}{12} = 0.5$, וזה אכן אופטימלי.

בעיה. אנחנו לא יודעים בזמן אמת מה הדף שנשתמש בו הכי פחות זמן בעתיד.

18.2 אלגוריתם אופטימלי (FIFO)

דוגמה. נסתכל על אותה מערכת:

$$0, 2, 1, 6, 4, 0, 1, 0, 3, 1, 2, 1$$

תחילה נקבל זכרון כך:

$$\boxed{\begin{matrix}0\\2\\1\\6\end{matrix}} \xrightarrow{0\to4} \boxed{\begin{matrix}4\\2\\1\\6\end{matrix}} \xrightarrow{2\to0} \boxed{\begin{matrix}4\\0\\1\\6\end{matrix}} \xrightarrow{1\to3} \boxed{\begin{matrix}4\\0\\3\\6\end{matrix}} \xrightarrow{6\to1} \boxed{\begin{matrix}4\\0\\3\\1\end{matrix}} \xrightarrow{4\to2} \boxed{\begin{matrix}2\\0\\1\\4\end{matrix}}$$

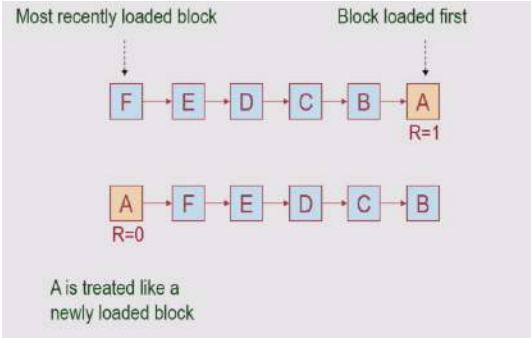
כמות הפעמים שהיה Page Fault היא 9 ולכן השגיאה היא $\frac{|\text{Page Faults Amount}|}{|\text{Pages Amount}|} = \frac{9}{12} = 0.75$, שזה פחות טוב ב-0.25 מהאופטימלי.

בעיה. מדיניות ההחלפה שלו לא הוגנת - סביר שמי שנמצא הכי הרבה זמן ב-RAM הוא הדף שמשתמשים בו הכי הרבה.

18.2.1 Second Chance FIFO (Clock Algorithm)

נתקן את FIFO על ידי הוספה של Reference Bit שנסמנו R. נפעל בדיוק כמו ב-FIFO, רק שאם $R = 1$ ניתן לדף הזדמנות שנייה, נגדיר $R = 0$, נזיז אותו לסוף התור ולא נחליף אותו.

באיור הבא נקבל המחשה:



איור 91: ב-FIFO הרגיל היינו פשוט מוציאים את A, אך כאן נתנו לדף הזדמנות שנייה:
הפכנו את הביט שלו להיות 0 והעברנו אותו לסוף התור. הפיכת הביט היא על מנת שהוא אכן יקבל הזדמנות שנייה בלבד, והחזרה לסוף התור, היא על מנת שלא נבחר בו פעם נוספת מיד לאחר מכן.

18.3 LRU (Least Frequently Used)

בכל שלב נבחר את הדף שלא השתמשנו בו הכי הרבה עד עכשיו.

דוגמה. נסתכל על הדפים שצריכים בסדר הבא:

$$0, 2, 1, 6, 4, 0, 1, 0, 3, 1, 2, 1$$

ונניח כי יש לנו 4 מסגרות. תחילה נקבל זכרון כך:

$$\boxed{\begin{matrix}0\\2\\1\\6\end{matrix}} \xrightarrow{0\to4} \boxed{\begin{matrix}4\\2\\1\\6\end{matrix}} \xrightarrow{2\to0} \boxed{\begin{matrix}4\\0\\1\\6\end{matrix}} \xrightarrow{6\to3} \boxed{\begin{matrix}4\\0\\1\\3\end{matrix}} \xrightarrow{1\to2} \boxed{\begin{matrix}4\\0\\2\\3\end{matrix}}$$

כמות הפעמים שהיה Page Fault היא 8 ולכן השגיאה היא $\frac{|\text{Page Faults Amount}|}{|\text{Pages Amount}|} = \frac{8}{12} = 0.67$, **וזה טוב יותר!**

בעיה. כיצד נוכל לשמור מידע על גישות לדפים? זה הרי דורש הרבה גישות זכרון בכל גישה לדף.

הפתרונות הם:

- שימוש במונים (Counters):
  - ◁ נוסיף לחומרה שעון שמתקתק בכל גישת זכרון.
  - ◁ כאשר ניגשים לדף, נסמן לו את הזמן של השעון.
  - ◁ נבחר את הדף עם הזמן המינימלי.
- שימוש במחסנית:
  - ◁ נשמור מחסנית עם גישות לדפים.
  - ◁ בכל גישה לדף, נעביר אותה לראש המחסנית.
  - ◁ נבחר את הדף שנמצא בסוף המחסנית.

בעיה. ההוספה של הרכיבים האלה לחומרה היא מאוד מורכבת, ולכן בפועל, לא משתמשים בו.

במקומו, משתמשים בשיטה הנקראת Pseudo LRU, שראינו בהרצאה.

18.4 שאלות חזרה - תרגום כתובות

שאלה נניח כי יש לנו מכונה עם זכרון פיסי 8GB, גודל דף 8KB, ונניח כי כל כניסה בטבלת הדפים היא בגודל 4Bytes. כמה רמות של טבלת דפים דרושות על מנת למפות כתובת וירטואלית בגודל 46Bit, אם ידוע שכל טבלת דפים נכנסת (בדיוק) בדף אחד.

פתרון. נמיר את הנתונים ליחידות של בתים:

$$\begin{aligned}\text{Ram} &= 2^{33}\text{Bytes}\\ \text{Page} &= 2^{13}\text{Bytes}\\ \text{TableEntry} &= 2^{2}\text{Bytes}\end{aligned}$$

נבצע חלוקה של הכתובת ל-$\mathrm{p_1 \mid\mid p_2 \mid\mid \ldots \mid\mid p_n \mid\mid offset}$. נבחין כי ה-Offset הינו בטווח $0, \ldots, 2^{13}-1$, ולכן אנו נשארים עם $p_1 \mid\mid p_2 \mid\mid \ldots \mid\mid p_n$ שתופסים 33 ביטים. מהנתון, הגודל של טבלה הוא $\text{Entries} \cdot 4\text{Bytes} = \text{Page Size} = 2^{13}\text{Bytes}$ ולכן מספר הכניסות הן 2^{11}Bytes, מה שאומר שכל טבלה דורשת מרחב הכתובות של 11 ביטים. מכאן, דרושות $\frac{33}{11} = 3$ טבלות כדי לבצע את החלוקה.

אבל, לא כמות הביטים המוקצה לטבלות, מתחלקת במספר הביטים הדרושים לטבלה. במקרה זה, מי שיכיל פחות ביטים, היא הרמה הראשונה, כלומר זו הנגשת לכל הטבלות. מדוע? שכן היא צריכה לגשת למספר מצומצם מאוד של טבלות, אז אם נקטין במעט את מרחב הכתובות שלה, זה לא ישנה כלום. למשל, אם גודל הכתובת הוירטואלית הוא 40Bit, אז היו לנו

$$\left\lfloor \frac{40-13}{11} \right\rfloor = \left\lfloor \frac{27}{11} \right\rfloor = 2$$

טבלות בגודל 11 ביט ועוד טבלה עליונה בגודל $(40-13) \mod 11 = (40-13) - \left\lfloor \frac{40-13}{11} \right\rfloor \cdot 11 = 27 - 22 = 5$.

שאלה בהמשך לשאלה הקודמת, פרטו את תוכן הכניסה בטבלה, כלומר את ה-PTE.

פתרון. כל כניסה היא בגודל 4Bytes. אנו יודעים שכניסה מכילה את מספר ה-Frame שבו מאוחסנת הטבלה עליה התא מצביע. היות שגודל מסגרת שווה לגודל דף, גודל המסגרת הוא 2^{13}Bytes, היות שהזכרון הפיסי הוא 2^{33}Bytes יש לנו $\text{Frames } \frac{2^{33}}{2^{13}} = 2^{20}$. על כן, דרושים 20 ביטים בשביל זה. נשארו לנו 12 ביטים. שם נשמור הרשאות קריאה, כתיבה, הרצה, שימוש וכו'.

שאלה בהמשך לשאלה, כמה גישות לזכרון דרושות כדי לכתוב או לקרוא מילה אחת בגודל 32Bit? הניחו שאין שימוש ב-Cache.

פתרון. מתקבלת כתובת וירטאולית בגודל 46Bit, הגישה הראשונה היא לטבלה הראשונה, אחריה לטבלה השנייה, השלישית, ולבסוף, למקום בזכרון אליו נרצה לקרוא או לכתוב. סך הכל **לפחות** 4 גישות שזה באופן כללי $\text{Levels} + 1$. **(לפחות)** כי יתכן שהדף הועבר לדיסק כדי לתת למקום לדפים אחרים. במקרה זה, נדרש להביא את הדף מהדיסק, ולכן מעבר לגישה לדף, נצטרך למצוא Frame ריק, או לעשות Evict ל-Page מה שדורש גישות נוספות.

שאלה כמה זכרון פיסי צריך תהליך שלו שלושה דפים של זכרון וירטואלי? (למשל סגמנט קוד, מידע, ומחסנית).

פתרון. שלושה דפים דורשים $3 \cdot 2^{13}$ בתים. אך תהליך צריך גם טבלת דפים, ולכן הוא יזדקק לדפים נוספים עבור הטבלה. נבחין אבל כי מספיקות שלוש טבלות בלבד, אחת ברמה הראשונה, שהיא תמיד תהיה, ולה כניסה אחת לא ריקה, שתצביע על טבלה שגם לה כניסה אחת מלאה, שתצביע על טבלה אחת ברמה השלישית, שלה שלוש כניסות, כל אחת עבור דף אחר. סך הכל $6 = 6 \cdot 2^{13}$ דפים.

18.5 שאלות חזרה - אלגוריתמי החלפה

התבונן בקטע קוד הבא שמאפס מערך של מספרים מסוג integer (כל integer הינו בגודל 4 בתים).

```
for (int i=0; i<2^29; i++) {
  numbers[i] = 0;
}
```

הנחות:

- למחשב זיכרון פיזי בגודל 2^{32} בתים המחולק למסגרות בגודל 2^{12} בתים. שימו לב: כל מסגרת מכילה עד 2^{10} איברים מסוג integer.
- איברי המערך מוקצים בצורה רציפה בתוך כל דף ומתחילים מתחילת הדף הראשון.
- הקוד כולו נכנס בדף אחד.
- שיטת החלפת דפים הינה demand paging ו- i נמצא ברגיסטר.
- בהתחלה הזיכרון מכיל רק את טבלאות הדפים והן נשארות תמיד בזיכרון (התעלמו מ-page faults על טבלאות הדפים).

א. (6 נק') נניח שהחלפת דפים מתבצעת במדיניות LRU. כמה page faults יהיו במהלך האלגוריתם אם מוקצים לתהליך 2^{12} מסגרות בזיכרון (לא כולל טבלאות דפים)? נמק.

איור 92 : שאלה

פתרון. מתקיים כי

$$\begin{aligned}
&\text{Ram} = 2^{32}\text{B} = 4\text{GB}\\
&\text{Page} = \text{Frame} = 2^{12} = 2\text{KB}\\
&\text{sizeof}\,(\text{int}) = 4\text{B}
\end{aligned}$$

מכאן אנו מקבלים כי יש $\frac{2^{32}}{2^{12}} = 2^{20}$ מסגרות בזכרון הפיסי ו-2^{12} מסגרות לתהליך. נסתכל על כמה מסגרות ניגשים במהלך הלולאה. ראשית נבחין כי דרושים 2^{29} מקומות במערך, שהם מתפרשים על

$$\frac{2^{29}\cdot 4}{2^{12}} = 2^{19}$$

מסגרות. היות שעוברים על המערך באופן סדרתי, מספיק רק דף אחד של המערך בכל פעם. נזכור שהתהליך צריך דף עבור הקוד שלו, ולכן בכל זמן נתון התהליך צריך 2 מסגרות זמינות. היות שהמדיניות היא LRU, דף הקוד לא יוחלף לעולם, ולכן היות שיש לנו 2^{12} מסגרות, יש לנו לפחות דף אחד עבור המערך בכל פעם, ולכן יהיה לנו 2^{19} Page Faults עבור המערך ועוד 1 עבור הקוד. כמו כן, 2 מסגרות היו מספיקות. סך הכל

$$\boxed{2^{19}+1}$$

שאלה בנתוני השאלה קודמת, נניח כעת שהחלפת הדפים מתבצעת במדיניות Second Chance FIFO. כמה Page Faults יהיו במהלך האלגוריתם אם מוקצים לתהליך 2^{12} מסגרות בזכרון. (לא כולל טבלות דפים).

פתרון. נבחין כי נקבל לפחות $2^{19}+1$ Page Faults. שכן כל ה-Page Faults שהיו קודם, יהיו גם עכשיו. מה שישנה את התשובה יהיה ה-CodePage. נסתכל על המקרה של FIFO רגיל. במקרה זה, כל 2^{12} PageFaults היה מתווסף Fault נוסף של ה-Page, כי הוא היה הבא בתור. אבל במקרה שלנו, יש לו הזדמנות שנייה. נבחין כי לכולם יש הזדמנות שנייה, ולכן, כאשר הביטים הם 1, כולם יועברו לסוף התור, כך שהוא יגיע להתחלה, ובמקרה זה נקבל בדיוק את המקרה של FIFO הרגיל, שכן הביט הוא 0. לכן נקבל עוד

$$\left\lceil \frac{2^{19}}{2^{12}} \right\rceil = 2^7$$

PageFaults עבור הקוד, וסך הכל

$$\boxed{2^{19}+1+2^7}$$

הסיבה שקיבלנו את אותו מקרה, היא ש**כל הדפים** מקבלים הזדמנות שנייה.

שאלה נתונה בחירת הדפים

$$1, 2, 3, 4, 1, 3, 2, 1, 5, 2, 3, 6, 5, 3, 2, 1$$

ויש 4 מסגרות. מי יהיו שלושת הקורבנות הראשונים באלגוריתם ה-Second Chance FIFO?

פתרון. נרשום כל דף יחד עם הביט שלו. נבחין כי כאשר רוצים דף, הביט שלו הופך ל-1.

$$\begin{bmatrix}1&1\\1&2\\1&3\\1&4\end{bmatrix}\to\begin{bmatrix}0&1\\0&2\\0&3\\0&4\end{bmatrix}\xrightarrow{1\to5}\begin{bmatrix}0&2\\0&3\\0&4\\1&5\end{bmatrix}\to\begin{bmatrix}1&2\\1&3\\0&4\\1&5\end{bmatrix}\to\begin{bmatrix}0&4\\1&5\\0&2\\0&3\end{bmatrix}\xrightarrow{4\to6}\begin{bmatrix}1&5\\0&2\\0&3\\1&6\end{bmatrix}\to\begin{bmatrix}1&5\\1&2\\1&3\\1&6\end{bmatrix}\to\begin{bmatrix}0&5\\0&2\\0&3\\0&6\end{bmatrix}\xrightarrow{5\to1}\begin{bmatrix}0&2\\0&3\\0&6\\1&1\end{bmatrix}$$

וניתן לראות שהקורבנות הם $1, 4, 5$.

18.6 חזרה - Fork

שאלה התבוננו בקטע הקוד הבא בו כל מילה בגודל 32Bit:

```c
int main(void)
{
        int var = 0;
        int i = 0;

        printf("%d %p\n", var, &var);

        for (i = 0; i < 3; ++i){
                if (fork() ==0){
                        var = var*2;
                        if (i == 2){
                                printf("%d %p\n", var, &var);
                        }
                } else {
                                var = 2*var + 1;
                                if (i == 2){
                                        printf("%d %p\n", var, &var);
                                }
                }
        }
        return 0;
}
```

נניח כי פקודות printf, fork אינן נכשלות והשורה הראשונה שהודפסה בפלט היא EA655A78 0. נזכיר כי fork יוצרת תהליך חדש שהוא העתק של התהליך שקרא לה, וכי היא מחזירה 0 עבור התהליך שנוצר, ואת ה-pid של התהליך שנוצר עבור האב. כתבו פלט אפשרי (השורות שהודפסו על ידי printf בסיום הריצה).

פתרון. נבין מה קורה כאן. תחילה התהליך הראשון רץ וקורא ל-fork. לכן נוצר לו תהליך חדש עם $i = 0$. הרבה תהליכים יווצרו ולכן נבחר באפשרות הפשוטה, לפי סדר היצירה, שכן בפועל יתכנו context switch שישנו את הסדר.

נבחין כי עבור כל תהליך מרחב הכתובות זהה, ולכן הכתובת של var תשאר זהה, כלומר EA655A78. לכן הדבר היחיד שישתנה הוא הערך של var.

נסתכל אם כך על עץ התהליכים:

- האבא $\left(\begin{smallmatrix}\text{var}=0\\ \text{i}=0\end{smallmatrix}\right)$
  - ◁ ילד ראשון, עם העתק $\left(\begin{smallmatrix}\text{var}=0\\ \text{i}=0\end{smallmatrix}\right)$, אבל נניח שממשיכים עם הבן, שנכנס ל-if הראשון, כי $\text{fork} == 0$. הוא לא מדפיס כלום כי $i \neq 2$, ולכן ממשיך לאיטרציה הבאה עם $i = 1$, ואז יוצר תהליך נוסף.
  - ◁ ילד ראשון עם העתק $\left(\begin{smallmatrix}\text{var}=0\\ \text{i}=1\end{smallmatrix}\right)$, כמו קודם, הוא נכנס לתנאי, כי $\text{fork} == 0$, אבל לא מדפיס כלום. ממשיכים לאיטרציה הבאה שם $i = 2$, ונוצר תהליך חדש.
    - ⋆ ילד ראשון עם העתק $\left(\begin{smallmatrix}\text{var}=0\\ \text{i}=2\end{smallmatrix}\right)$. ילד זה נכנס ל-if ומדפיס EA655A78 0, ומסיים מיד לאחר מכן, מבלי ליצור עוד תהליך.
  - ◁ חזרה לילד הראשון $\left(\begin{smallmatrix}\text{var}=0\\ \text{i}=2\end{smallmatrix}\right)$, הוא הולך ל-else שכן ביצירת התהליך קודם הוא לא קיבל 0, ולכן הוא יגדיל את var להיות 1 וידפיס EA655A78 1, מיד לאחר מכן יסיים את ריצתו.

◁ חזרה לילד הראשון $\binom{\text{var}=0}{\text{i}=1}$ הוא ילך ל-else ויקבל $\text{var} = 1$, אבל $i = 1$ ולכן לא ידפיס כלום, מכאן מקבלים חזרה על התהליך הקודם, רק שהפעם כל הדפסה תעלה ב-1, עד כדי כמה הדפסות שלא יתרחשו כי $i = 1$ ולא 0.

אפשר להמשיך לבנות את העץ, אך בפעול מתקיים כי זה עץ בינארי עם 3 רמות, ולכן נקבל $2^3 = 8$ הדפסות, כל אחד עם var שונה. לכן סך הכל

0 EA655A78
1 EA655A78
2 EA655A78
3 EA655A78
4 EA655A78
5 EA655A78
6 EA655A78
7 EA655A78

אבל, היות שהסדר יכול להיות כל פרמוטציה של הדפסות אלה, מה שנקבל זה $8!$ הדפסות שונות אפשריות, כל הסידורים האפשריים, כאשר כל תהליך מדפיס את אותה הדפסה, באופן דטרמינסטי.

שאלה בהמשך לשאלה הקודמת, מהו מרחב הכתובות הוירטואליות המוקצה לכל תהליך ב-words? הסבירו. אם לא ניתן לדעת, ציינו זאת והסבירו מדוע. מה גודל הזכרון הפיסי?

פתרון. נבחין כי כתובת וירטואלית היא מהצורה EA655A78. נבחין כי כל ספרה דורשת 4 ביטים לכן הכתובת דורשת $4 \cdot 8 = 32$ ביט, מכאן יש 2^{32} כתובות. היות שמילה היא $32\text{Bit} = 2^2\text{B}$ נקבל שזה 2^{30}Words.

לא ניתן לדעת מהו הזכרון הפיסי שכן לא ניתן לנו שום מידע עליו.

19 תרגול 9 - מערכות קבצים (File Systems)

קבצים

כבר הכרנו את הזכרון **הלא-נדיף** בהקשר של Paging, אך עכשיו נתייחס לשימוש היותר-מוכר שלו: מערכות קבצים.

תזכורת משמעות הביטוי לא-נדיף הוא שכאשר המחשב מתכבה, הנתונים ישמרו גם אחרי שנדליק אותו שוב. גם כאשר תהליך אחד משתמש בקובץ ומסיים את ריצתו, תהליך אחר יכול להשתמש באותו קובץ (ולראות גם מה התהליך הראשון כתב).

אז מהו **קובץ**?

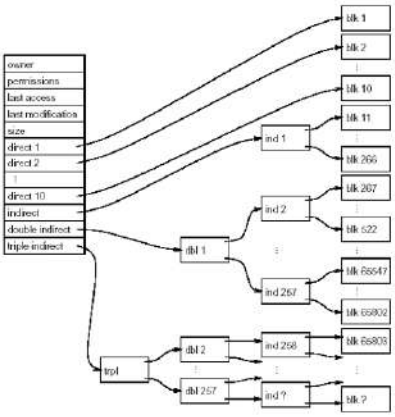
- קובץ הוא יחידת מידע (רצף בתים), **לא-נדיפה**, ולו יש כמה תכונות:
  - ◁ שם הקובץ
  - ◁ התוכן שלו
  - ◁ מטא-דאטא
    - ◁ גודל, בעלים, מיקום בדיסק, הרשאות (מי יכול לכתוב/לקרוא/להריץ), סוג קובץ (בינארי/טקסט או קובץ רגיל/תיקייה), תאריך יצירה/שינוי אחרון וכו'
    - ◁ יש כל מיני פעולות שאפשר להפעיל על קובץ (קריאה, כתיבה וכו')

הערה שם הקובץ לא נמצא במטא-דאטא של הקובץ, אלא בתיקייה שמחזיקה את הקובץ (נראה זאת בהמשך).

במערכות Unix, המידע על הקובץ שראינו ברשימה הקודמת מאוחסן בתוך אובייקט מיוחד בשם inode. ה-inode לא מכיל את התוכן של הקובץ עצמו (וכאמור גם לא את שם הקובץ), אלא את המידע עליו, כולל איפה תוכן הקובץ עצמו נמצא: המיקום בדיסק.

איך מיקום הקובץ שמור? הקובץ לא בהכרח שמור כולו כרצף אחד ארוך, אז כמו בניהול זכרון, בו השתמשנו ב-page-ים, כאן אנו משתמשים בבלוקים. ה-inode של כל קובץ מכיל מצביעים לבלוקים המתאימים לקובץ הבנויים באופן היררכי כדי לייעל את השימוש באחסון:

- מצביעים ישירים:
  - ◁ מצביע ישירות על בלוק של הקובץ.
    יש מצביעים בודדים כאלה (~ 12) המאפשרים לשמור על 48KB באחסון.
- מצביעים עקיפים:
  - ◁ מצביע על בלוק שמכיל מצביעים המצביעים על בלוקים של הקובץ. כך באמצעות מצביע יחיד השמור ב-inode אנו יכולים להצביע על המון בלוקים.
    יש מצביע יחיד כזה המכיל 1024 מצביעים נוספים המאפשרים לשמור על 4MB נוספים באחסון.
- מצביע עקיף כפול:
  - ◁ מצביע על בלוק שמכיל מצביעים על בלוקים שמכילים מצביעים על בלוקים של הקובץ. כך באמצעות מצביע יחיד השמור ב-inode אנו יכולים להצביע על אפילו יותר בלוקים.
    יש מצביע יחיד כזה המאפשר לשמור על $4GB$ נוספים באחסון.
- מצביע עקיף משולש:
  - ◁ אותו סיפור כמו קודם, רק עם רמה נוסף של בלוקים (ראו איור).
    יש מצביע יחיד כזה המאפשר לשמור על $4TB$ נוספים באחסון.



איור 93: מבנה ה-inode. שימו לב להיררכיית המצביעים (מהישירים לעקיפים)

תיקיות

תיקיות הן גם קבצים, אך במטא-דאטא של ה-inode רשום שהסוג שלהן הוא תיקייה.

בניגוד לקובץ רגיל, בו אנו יכולים לשמור דברים כמו מכתבי אהבה או סרנדות, בתיקייה נמצאים המיפויים של הקבצים בתוך התיקייה.
כלומר, עבור כל קובץ אנו רושמים בתוכן התיקייה את שם הקובץ ואת מספר ה-inode של אותו הקובץ.

הערה תוכן התיקייה (כלומר המיפוי הזה) נקרא **מדריך**.

19.1 Superblock

נשאלת השאלה: איך אנו מקצים בלוקים עבור קבצים (ותיקיות)?

מערכת ההפעלה שומרת אובייקט ב-RAM (באזור ששייך למערכת ההפעלה) בשם Superblock.

ה-Superblock שומר מידע על מערכת הקבצים כולל:

- גודל מערכת הקבצים.
- רשימת בלוקים פנויים במערכת הקבצים.
- רשימת inode-ים פנויים שאפשר להשתמש בהם.
- ועוד.

עכשיו עם ה-Superblock והמידע השמור בו אנחנו יכולים להוסיף מידע לקובץ קיים או ליצור קובץ חדש.

הקצאת inodes

כמות ה-inode-ים במערכת הקבצים היא סופית: ה-inode-ים נוצרים פעם אחת כשמערכת הקבצים נוצרת בפעם הראשונה. כל ה-inode-ים יושבים באזור מיוחד בדיסק. ניתן לדמיין שמספר ה-inode של הקובץ למעשה מתאר את האינדקס של ה-inode במערך ה-inode-ים.
כלומר, תיאורטית יכול להיות שתיצרו כל כך הרבה קבצים, שיגמרו כל ה-inode-ים ולא תוכלו ליצור עוד קבצים. אולם הכמות כה גדולה שבפועל סביר להניח שזה לא קרה לאף אדם שאתם מכירים.

שאלה למה לא ליצור inode-ים דינאמית?

תשובה לפי עידן רפאלי נדמה שזה עניין של יעילות (זה סטטי לצורכי אופטימיזציה וייעול).

בגלל שבדר"כ יש המון המון inode-ים פנויים, במקום לשמור את כולם, ה-Superblock שומר (מעין cache) רק רשימה קצרה ככה שברגע שנרצה ליצור קובץ נוכל ישר לשלוף inode מהרשימה.
כאשר inode משוחרר (כלומר מחקנו קובץ) מוסיפים אותו לרשימת ה-inode-ים הפנויים. כאשר רשימת ה-inode-ים הפנויים ב-Superblock מתרוקנת, מערכת ההפעלה מחפשת בדיסק אילו inode-ים פנויים ומוסיפה כמה לרשימה שב-Superblock.

הקצאת בלוקים

באותו אופן, מערכת ההפעלה שומרת רשימה של **בלוקים** פנויים בדיסק בהם היא יכולה להשתמש כאשר אנו רוצים לכתוב לקובץ. ברגע שאנו מוחקים קובץ או מקטינים את כמות המידע בו, ומשתחרר בלוק, אנו מוסיפים אותו לרשימת הבלוקים הפנויים. ושוב, כאשר ברשימה של מערכת ההפעלה אין יותר בלוקים פנויים, הולכים לדיסק לחפש בלוקים כאלה.

עבודה עם קבצים כוללת הקצאת בלוקים עבור הקובץ, יחד עם פעולות קריאה/כתיבה. המטרה היא למזער גישות לדיסק (כי הוא מאוד איטי ביחס לשאר הרכיבים), לכן הרבה פעמים אנו נרצה לשמור את הבלוקים בזכרון, ולכתוב אותם בדיסק רק בשלב יותר מאוחר.
כך במקום לכתוב לדיסק שוב ושוב, אנו נכתוב את התוצאה הסופית פעם אחת ונחסוך את כל גישות הביניים.

הערה. למי מכם קרה שלאחר ניתוק ה-diskonkey מהמחשב, הקובץ שהעברנו אליו לא נשמר. זה לא באג ב-diskonkey, זה קורה כי המידע לא באמת עבר לדיסק, אלא נשאר ב-buffercache של מערכת הקבצים של ה-diskonkey, כי היא לא הספיקה להעתיק אותו.

פתיחת קובץ

כאשר אנו פותחים קובץ לשם קריאתו, לדוגמה באמצעות

$$\text{fd} = \text{open}\ (\text{"myfile"}, \text{R})$$

אנו מקבלים כערך החזרה File Descriptor (FD). זהו מספר שבאמצעותו (ורק באמצעותו) נוכל לבצע פעולות על הקובץ. בפועל המספר הזה הוא אינדקס בטבלת קבצים שמערכת ההפעלה שומרת לכל תהליך: טבלת ה-File Descriptors **שייחודית לכל תהליך**.

הערה Thread-ים חולקים את טבלת ה-File Descriptors כי הם תחת אותו תהליך.

כל פעם שתהליך נוצר, נוצרים עבורו שלושה קבצים מיוחדים, עם File Descriptor-ים מוגדרים מראש. אולי תופתעו לשמוע שמדובר בקבצים, אך כפי שאומרת האמירה המפורסם: "בלינוקס כל דבר הוא קובץ!".

- (stdin) Standard Input - $FS = 0$. מקבל קלט מהמשתמש.
- (stdout) Standard Output - $FS = 1$. מדפיס תוכן על המסך.
- (stderr) Standard Error - $FS = 2$. מדפיס שגיאות על המסך.

אז מה קורה באמת כשאנחנו מבצעים את הפקודה:

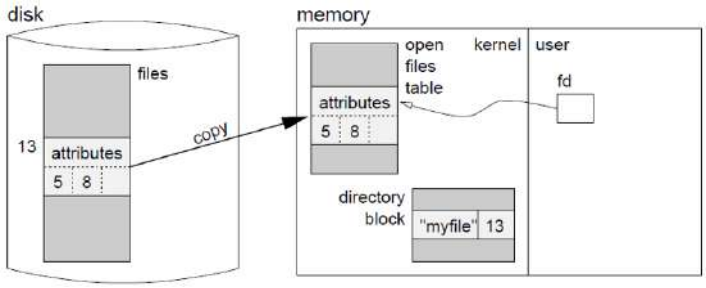
$$\text{fd} = \text{open}\,(\text{"myfile"}, \text{R})$$

בגלל שלא נתנו נתיב מלא, ההנחה היא שאנו מתכוונים ל-"myfile" בתיקייה הנוכחית. לכן אנו קוראים את המדריך של התיקייה הנוכחית ומוצאים שמספר ה-inode של "myfile" הוא מספר 13.

אנו הולכים לדיסק לאינדקס 13 ברשימת ה-inode-ים, קוראים את ה-inode וטוענים אותו לזכרון לטבלת ה-$open - files$ (טבלה זו מכילה את כל הקבצים שנפתחו אי פעם לכל התהליכים ביחד, היא **משותפת לכולם ולכן יחידה**).
לבסוף מוחזר לתוך המשתנה fd האינדקס בטבלת ה-File Descriptors, שם יש מצביע על המקום בטבלת ה-$open - files$ בו נמצא ה-inode של הקובץ.

שאלה למה צריך את טבלת ה-$open - files$?

תשובה בטבלה זו אנו יכולים לשמור הרשאות ואת ה-offset של הקובץ (המיקום ממנו אנו רוצים להתחיל לקרוא/לכתוב).



איור 94: סכימת פתיחת קובץ

קריאת קובץ

אם נרצה לקרוא את 100 הבתים הראשונים של הקובץ לתוך מערך buf, נוכל להשתמש בפקודה

$$\text{read}\,(\text{fd}, \text{buf}, 100)$$

אך מה באמת קורה כשמשתמשים בפקודה זו?

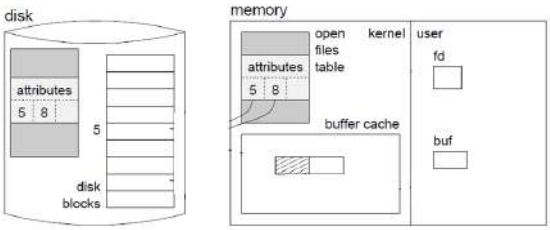
ה-fd מצביע על המיקום בו נמצא הקובץ בטבלת ה-File Descriptors (שמבציע בתורו על טבלת ה-$open - files$ בה נמצא ה-inode של הקובץ וה-offset ממנו יש להתחיל לקרוא).
בתוך ה-inode נמצאים המצביעים לבלוקים בדיסק בהם תוכן הקובץ שמור. בפועל לא צריך לקרוא את כל הבלוקים, אלא רק את אלה בהם נמצאים 100 הבתים הראשונים שביקשנו.
אנו טוענים לתוך ה-buffer cache את הבלוקים הרלוונטיים.
לתוך buf אנו מעתיקים את המידע הרלוונטי (לתוך ה-buffer cache אנו מעתיקים בלוקים שלמים בעוד של-buf אנו רוצים להעתיק רק 100 בתים ספציפיים!)

הערה לאחר שנקרא את 100 הבתים, ה-offset יתעדכן ויצביע על המקום ה-100 במקום ה-0. אם ברצוננו עכשיו לקרוא את הבית ה-5000, אנו יכולים להשתמש בפקודות כמו seek כדי לשנות את ה-offset.

שאלה למה טוענים ל-buffer cache בלוקים שלמים אם אנחנו רוצים פחות?

תשובה סביר להניח שאם כתבנו/קראנו מבלוק כלשהו, אנו נרצה לעשות זאת שוב לאותם בתים או לבתים סמוכים. לכן נחסוך קריאה מהדיסק ונשמור ישר את הבלוק בתוך ה-buffer cache. חוץ מזה, בלוקים זו יחידת המידע הבסיסית בה מועתקים דברים.

הערה שימוש ב-buffer cache מייעל את הכתיבה/קריאה, אבל גם מוביל לבעיות אמינות: בגלל שאנחנו לא כותבים את השינויים שביצענו לדיסק ישר כשביצענו אותם (אלא רק שומרים אותם ב-buffer cache כדי לכתוב אחר כך), אם המחשב יתכבה בפתאומיות, השינויים שלנו עלולים לא להישמר.
ודאי קרה לכם פעם ששכחתם לעשות *save* לקובץ ב-word ואיבדתם חלק ממה שכתבתם, או שניתקתם דיסק און-קי בפתאומיות (בלי לעשות "הוצאה בטוחה") והשינויים שביצעתם לא נשמרו.



איור 95: סכימת קריאת קובץ

כתיבת קובץ

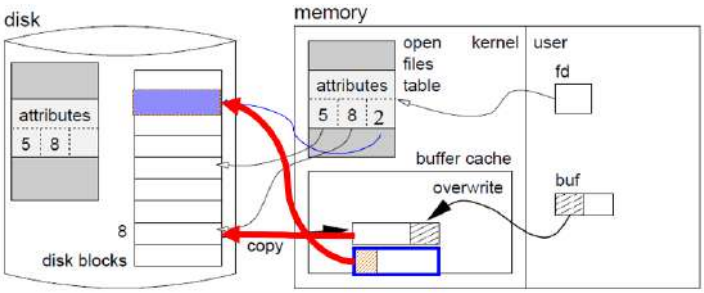
נניח שאנו רוצים לכתוב 100 בתים החל מהבית ה-2000 כאשר גודל כל בלוק הוא 1024.

זה אומר שאנחנו רוצים לכתוב לסוף הבלוק השני ותחילת הבלוק השלישי.

מסיבה זו, אנו נעתיק ל-buffer cache את שני הבלוקים הללו, ונשנה את הבתים הרצויים.

אם אין בלוק שלישי, אלא רק שני בלוקים, אנו צריכים להקצות בלוק חדש מבין רשימת הבלוקים הפנויים.
עם זאת, בגלל שמדובר בבלוק חדש אין צורך לקרוא את התוכן שלו מהדיסק (אין שם תוכן משמעותי בכלל, הוא רק הרגע הוקצה) ולכן פשוט נקצה בלוק בתוך ה-buffer cache ונחסוך קריאה לדיסק.

לסיום, ניקח את הבלוקים שכתבנו אליהם ב-buffer cache ונעתיק אותם לדיסק.

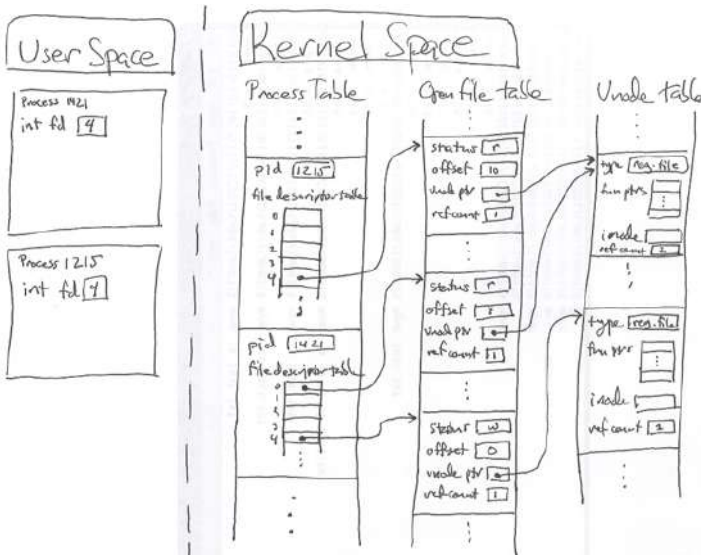


איור 96: סכימת כתיבת קובץ. בכחול נמצא הבלוק החדש שהקצנו (בלוק מספר 2), שמעולם לא קראנו אותו מהדיסק בניגוד ליתר הבלוקים כי הוא בלוק חדש.

סיכום טבלות קבצים

נעשה סדר בכל הטבלות השונות שראינו, כדי למנוע בלבול.

- טבלת ה-File Descriptor - טבלה שונה לכל תהליך. כל כניסה בטבלה מצביעה על כניסה בטבלת ה-Open Files. האינדקס בטבלת ה-File Descriptor הוא זה שנשמר ב-fd שמוחזר ע"י open.
- טבלת ה-Open Files - טבלה המשותפת לכל התהליכים הפתוחים. כל פעם שנפתח קובץ חדש, מוסיפים אותו לטבלה זו. כל כניסה בטבלה מצביעה על כניסה בטבלת ה-inode. מלבד זאת, בכל כניסה שמור גם ה-offset של הקובץ. כלומר, יכולים להיות כמה קבצים פתוחים המצביעים לאותו inode, ומה שיבדיל אותם יהיה ה-offset בתוך הקובץ.
- טבלת ה-inode - טבלה בה נמצאים ה-inode-ים של כל הקבצים. בטבלה זו כל קובץ יופיע לכל היותר פעם אחת (כי אין טעם לשמור את ה-inode כמה פעמים, זה סתם בזבוז מקום).



איור 97 : סכימת הטבלות השונות.

דוגמה. נניח שאנו רוצים לבצע את הפעולה

$$\text{fd} = \text{open}\,(\text{"/x/y/z/foo"}, \text{O\_CREATE})$$
$$\text{write}\,(\text{fd}, \&\text{buf}, 13)$$
$$\text{close}\,(\&\text{fd})$$

כמה פעולות של קריאה וכתיבה יתבצעו? הניחו ששום דבר לא נמצא מראש בזכרון, וכי הקובת קיים, אך ריק.

נבין קודם כל מה קורה כאן. תחילה ניגשים לתיקיית ה-root ומסתכלים ב-metadata שלה בתוך ה-inode, מה שדורש גישה לטבלת ה-inodes ולכן גישה לדיסק. לאחר מכן צריך לגשת למדריך של התיקייה, כלומר לקרוא ממנה את המידע על הקבצים, ואנו מבינים שצריך לחפש את הקובץ x בתוך התיקייה. לכן צריך לגשת שוב ל-inode ולמדריך שלו כדי לגשת לקובץ y. אנו רואים שלכל תיקייה דרושות שתי גישות - פתיחה, וקריאה, כלומר הקצאת inode בטבלה, וקריאתו.

יש לנו כאן 4 תיקיות ולכן 8 גישות. לאחר מכן, יש לנו עוד גישה ל-inode של foo, אך היות שהוא ריק, לא צריך עוד גישות, והכתיבת אליו בפונקציה write תתבצע מה-direct_pointer, מה שדורש גישה אחת לזכרון. עד כה $8 + 2 = 10$ גישות. לבסוף נסגור את הקובץ מה שיכתוב את הקובץ חזרה לדיסק, ולכן סך הכל 11 גישות לזכרון.

20 תרגול 10 - Containers and Networking

20.1 Containers

מכונה וירטואלית (VM) היא אמולציה של מערכת ההפעלה - אפשר ליצור כמה מהן במחשב אחד, והן מבודדות.
שימוש בהן דורש משאבים רבים. על כן אפשר להשתמש בגרסא חלשה יותר, שהיא Container:

- הוירטואליזיציה היא רק למערכת ההפעלה.
- **אין** וירטואליזציה לחומרה.
- אפליקציות שונות ירוצו מעל שכבת הקונטיינר, שיספק להן שירותים של מערכת הפעלה אחרת, כמו ספריות. בפועל הן רצות על מערכת ההפעלה **הראשית**.

(Linux Containers) יש בקונטיינרים של לינוקס שני רכיבים עיקריים:

- Namesapce - מה הקונטיינר רואה.
- cgroup - מה הקונטיינר יכול לעשות.

20.1.1 Namespace

מגדירים אילו רכיבי מערכת ההפעלה, תהליכים בתוך הקונטיינר יכולים לראות. למשל ה-Process IDs, שמות הבעלים, מזהי משתמשים, מערכות הקבצים (אילו קבצים הוא רואה, האם הוא רואה קבצים שלא נוצרו מהקונטיינר), תקשורת ועוד.

כשאומרים namespace הכוונה לרכיב אחד, או לכמה רכיבים.

תהליכים יכולים ליצור namespace חדש, ולהצטרף לאחד קיים. ככלל, תהליך אחד מתחיל ביצירת namespace ויוצר תהליכים שמצטרפים אליו, שכן התהליכים נוצרים באופן היררכי.

בפועל, ה-host של המערכת, מתחיל ביצירת namespace אחד לכל רכיב, שמשותף לכל התהליכים.

הדרך ליצור תהליך חדש עם namespace חדש, היא באמצעות הפונקציה clone שמבצעת system call:

```
int clone(int (*fn) (void *), void *stack, int flags, void *arg);
/**
        * stack - a pointer for the new process's stack
        * flags - a set of attributes for the new process, including namespaces
        * arg - the argument for fn.
        * returns the pid of the new child process
*/
```

לדוגמא, בתכנית הבאה:

```
#include <stdio.h>
#include <sched.h>

#define STACK 8192

int child(void *msg){
        printf("%s\n", (char*)msg);
        return 0;
}

int main(int argc, char* argv[]){
        void *stack = malloc(STACK); //  create stack
        char [] msg = "Hello from parent";
        int child_pid = clone(child, stack+STACK, CLONE_NEWPID | SIGCHLD, msg);
        // CLONE_NEWPID is a namespace for PID, that is, the child will have a new PID
            namespace.
```

```
        // SIGCHLD means that the child should send its father a signal, when it finishes.
        wait(NULL); //  waits for the child to finish, when finished, a signal is sent from the child,
        // the os will send the child, not the child itself.
        // That's why we set SIGCHLD
        return 0;
}
```

בדוגמא הזו, התהליך הבן יחשוב שהוא תהליך מספר 1, שכן יש לנו PID namespace משלו.

הרצת תכנית חדשה נניח שאנו בקונטיינר ואנו רוצים רוצים להריץ תכנית חדשה. יש קבוצה של פונקציות system calls שיכולות לעזור (exec*):

```
int execvp(const char *file, char *const argv[]);
/**
        * file - executable file to run
        * argv - array of arguments for the command, including the command itself as the first argument (!!!). It has to end with (char *)0
        * arg - the argument for fn.
*/
```

עם פונקציות אלה נוכל להריץ את קטע הקוד הבא מתוך הקונטיינר:

```
int main(int argc, char* argv[]){{
        char *_args[] = {"/bin/echo", argv + 1, (char*)0};
        int ret = execvp("/bin/echo", _args);
        return 0;
}
```

namespaces **מוכרים**

- CLONE_NEWUTS - מאפשר להגדיר HOSTNAME שונה מהאחד של ה-HOST. באופן דיפולטיבי, הבן מקבל את ה-Hostname של האבא, אבל אם הוא ישנה אותו, הוא לא ישנה אותו עבור האב.
- CLONE_NEWPID - מספק קבוצה בלתי תלויה של שמות של תהליכים (PIDs). התהליך הראשון ב-namespace יהיה עם $PID = 1$.
- CLONE_NEWNS - מספק קבוצה בלתי תלויה של מערכות קבצים שאליהן עשינו Mount. כמו במקרה הראשון, שינוי בתהליך הבן, לא ישפיע על תהליך האב.

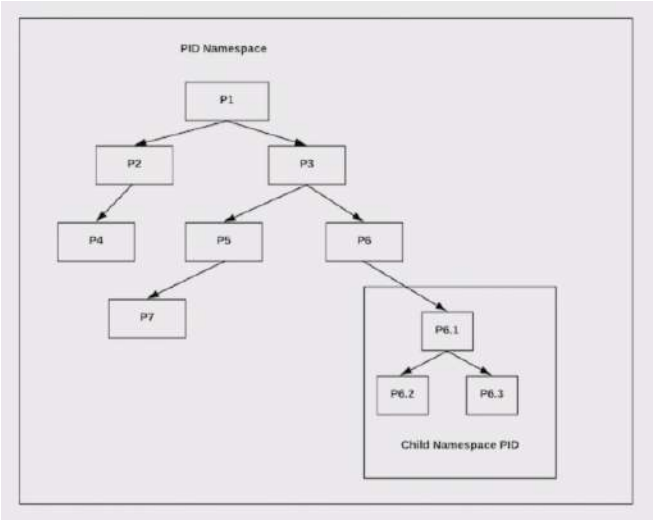
נניח אם כך שאנו רוצים לשנות את ה-Hostname של תהליך. נביט בתכנית הבאה:

```
int sethostname(char *name, size_t len);
/**
        * name - new hostname
        * len - size of name
*/

#include <stdio.h>
#include <sched.h>

#define STACK 8192
```

```
int child(void *arg){
        char *name = (char *) arg;
        sethostname(name, strlen(name));
        return 0;
}

int main(int argc, char* argv[]){
        void *stack = malloc(STACK); //  create stack
        char [] name = "Container";
        int child_pid = clone(child, stack+STACK, CLONE_NEWTS | SIGCHLD, name);
        wait(NULL);
        // the child will call the child function, with the hostname "Container"
        // The father's host name won't change
        return 0;
}
```

פשוט יצרנו תהליך חדש שמשנה לעצמו את ה-hostname. נעיר כי לא נוכל להריץ קוד זה על מחשבי האקווריום, כיוון שהפונקציה sethostname דורשת הרשאות גישה מיוחדות.

נבחין כי עבור תהליכים PID, התהליכים שהוא יוצר בעלי המזהים $PID.1, \ldots, PID.n$:



איור 98: כיצד תהליך רואה את המזהים של הילדים שלו. כל זה, כל עוד ה-namespace של ה-pid שונה.

20.1.2 יצירת file system חדש

שאלה כיצד נגדיר file system חדש?

הקונטיינר יצטרך לעבוד עם מערכת קבצים חדשה שהיא העתק של מערכת הקבצים של linux, כלומר ההעתק יכלול את הרשימה: bin, proc, dev, home וכדומה.

הוא לא יראה את הקבצים של תהליכים שרצים מחוצה לו.

מה שאומר שלאחר שניצור תהליך חדש עם CLINE_NEWNS, נצטרך לבצע לו mount למערכת הקבצים החדשה בתוך ה-container.

כדי לעשות זאת, עלינו לשנות את ה-root של התהליך להעתק של מערכת הקבצים שיצרנו, ולבצע mount לתיקייה proc, ואז כל התהליכים הילדים שיווצרו מה-container ישמרו בתיקיית ה-proc של הקונטיינר, ולא של תהליך האב שיצר אותם, כלומר של מערכת ההפעלה המקורית.

שאלה כיצד לשנות את השורש?

```
int chroot(const char *path);
```

```
/**
        * path - Path to the new root directory
*/
```

שאלה כיצד לבצע mount?

```
mount("proc", "/proc", "proc", 0, 0);
/ **
        * you can refer to it as a black box, since it is a bit complicated.
*/
```

כאמור, היופי בפקודה זו, הוא שלאחר הרצתה, הקבצים שיווצרו מהקונטיינר, ישמרו ב-proc שלו ולא של מערכת ההפעלה הראשית.

בסוף, צריך לבצע את הפקודה unmount.

20.1.3 cgroup

מגדירים מה הקונטיינר יכול לעשות. למשל, ניתן להגבל את כמות התהליכים בו ל-100, את כמות הזכרון ל-1GB, את ה-CPU ועוד.

כדי להגביל את מספר התהליכים, עלינו ליצור תיקייה חדשה בנתיב "sys/fs/cgroup/pids", תחת השורש החדש שהגדרנו קודם עבור הקונטיינר.

כדי ליצור את התיקייה נשתמש ב-() mkdir. התיקייה cgroup תיצור אוטומטית קבצים בתוך התיקייה, ברגע שהתהליך יווצר.

בתוך התיקייה הזו, יווצר הקובץ cgroup.procs. כדי להכניס תהליך לתוך ה-cgroup החדש, צריך לכתוב את ה-pid שלו אל תוך הקובץ, תחת התיקייה שיצרנו.

לאחר מכן, עלינו לכתוב אל pids.max את מספר התהליכים שאפשר ליצור. לבסוף, עלינו לשחרר את המשאבים של הקונטיינר, כאשר הוא מסיים. את זאת מבצעים על ידי כתיבה של "1" אל תוך הקובץ notify_on_release.

20.2 Networking

אנו מדברים על תקשורת בין רכיבים ברשת, בפרט בין מחשבים. לכן עלינו לדבר על פרוטוקולי תקשורת.

מהו פרוטוקול? זוהי שפה משותפת שמסכימים עליה שני צדדים. למשל, כאשר תהליך שולח אחד הודעה, התהליך שמקבל אותה צריך לדעת לפרש אותה.

20.2.1 Protocol Stack

נניח שאנו רוצים לשלוח אימייל. האימייל מכיל את הכתובת שלנו, כתובת של היעד והמידע. כיצד המידע עובר? נגיד, ברצף דרך ערוץ מידע?

על פניו זה נשמע סביר, הבעיה היא שמידע ארוך עלול להשתבש עקב הפיסיקה של העברת המידע, למשל ביטים מתהפכים וכדומה.

על כן, אנו שולחים את המידע בפקטות קטנות יותר, מצד לצד. יחד עם זאת, לאחר חלוקה לפקטות, בעיות חדשות צצות. למשל:

- פקטה אבודה.
- סדר הפקטות משתבש.

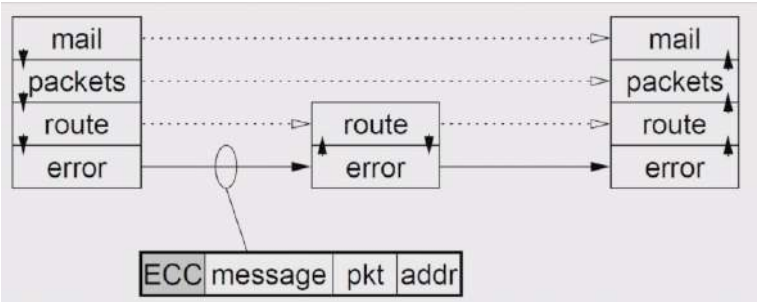
לכן, עלינו להוסיף מזהים כדי למנוע בעיות אלה, נגיד מספר סידורי לפקטה ועוד.

מלבד זאת, בין מחשבים ברוב המקרים אין כבל העברת מידע, ולכן צריך לנתב את הפקטות ברחבי הרשת. על כן, צריך לבצע routing בין הרכיבים השונים.

לכן, כל פקטה צריכה להכיל את כתובת היעד, וצופן תיקון שגיאות - בלוק ECC.

כל השלבים שתיארנו יצרו "מחסנית" של פרוטוקולים שונים, וכדי לקרוא את ההודעה המקורית, צריך לעבור על הפרוטוקלים במחסנית.

לסיכום זה נראה כך:



איור 99 : המחשה לפרוטוקול המחסנית ולמבנה ההודעה

מה שהצגנו כאן הוא הפשטה של מודל מורכב יותר, שנקרא מודל "5 השכבות" :

- השכבה הפיסית (Wifi, Ethernet, 802.11) :
  - ◁ פרוטוקולי תקשורת בין רכיבים המחוברים פיסית, כלומר הרכיבים מחוברים פיסית אחד לשני.
- שכבת הרשת (IP) :
  - ◁ העברת פקטות בין מקור ליעד דרך הרשת, ורשתות שונות, על ידי ניווט הפקטה.
- שכבת התעבורה (TCP, UDP) :
  - ◁ שירותי העברת מידע בין אפליקציות שונות - דאגה שההודעה תגיע בסדר הנכון.
  - ◁ נשים לב שכאן משתמשים בטרמינלוגיה של אפליקציות ולא של רכיבי רשת/נתבים וכו'.
- שכבת האפליקציה (HTTP/S, SSH, FTP, DNS) :
  - ◁ תקשורת בין תהליכים.

20.2.2 שכבת התעבורה

שכבת הרשת מעבירה כל פקטה מבלי לדעת על קיומה של פקטה נוספת שחולקת איתה הודעה גדולה יותר. לכן, יתכן שפקטות לא יגיעו לפי הסדר הנכון.
שכבת התעבורה נועדה לטפל בשיבושי "תנועה" אלה, במטרה לספק תקשורת "אמינה" בין אפליקציות.
לשכבה זו שני פרוטוקולים עיקריים TCP, UDP.

User Datagram Protocol (UDP) הפרוטוקול מוסיף לכל פקטה מידע נוסף והוא מספר הפורט הרלוונטי באפליקציה.
הוא מוסיף את אורך הפקטה, ו-checksum המאפשר לבצע ואלידציה. הפרוטוקול לא אמין לחלוטין, שכן יתכן שפקטה תאבד בדרך, תגיע בסדר לא נכון.
הוא לא מבוסס קישור, הוא פשוט שולח פקטה, היעד לא מצפה לקבל מהשולח מידע.
ה-Header מכיל את ה-

Source Port, Destination Port, Length, Checksum, Application Data

Transmission Control Protocal (TCP) זהו פרוטוקול מבוסס קישור שמבטיח את אמינות המידע. הוא מטפל במקרה של פקטות אבודות, הגעה בסדר לא נכון, כפילויות ומרווחי זמן.
נוכל להשוות בין שני הפרוטוקולים :

תכונה	UDP	TCP
אמינות	לא	כן
מבוסס קישור?	לא	כן
בקרת זרימה	לא	כן
מהירות	גבוהה	נמוכה
אפליקציות	VOIP, Most Games	HTTP, HTTPs, FTP, SMTP, Telnet, SSH

טבלה 2 : השוואה בין UDP, TCP

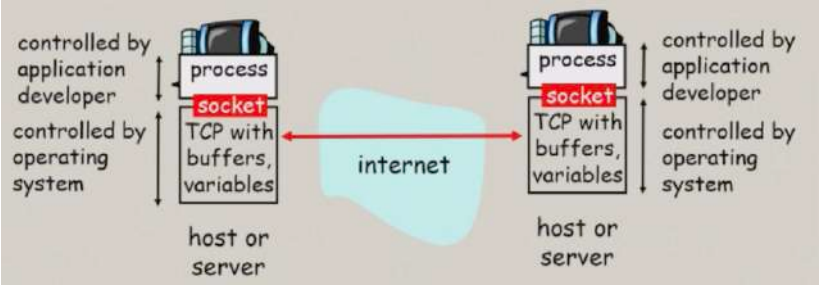
אלה פרוטוקלים חשובים מאוד בשכבת התעבורה. קיימים פרוטוקולי תקשורת רבים גם בשכבת הרשת, עליהם מומלץ לקרוא, בלי קשר לחומר הקורס.

21 תרגול 11 - Sockets

המטרה שלנו היא לבנות אפליקציה של שרת-לקוח שתוכל לבצע תקשורת באמצעות sockets.
האובייקט socket מאפשר חיבור TCP/UDP, ולכן נוח להשתמש בו.

21.1 Sockets בפרוטוקול TCP

כאשר תהליכים מנסים לתקשר ביניהם באמצעות sockets, באופן אמין, כלומר, מבוסס קישור, הם מגדירים את פרוטוקול התקשורת להיות TCP.
המידע שנשלח, נבחר על ידיהם, אך המימוש הפנימי של ה-sockets, כולל חלוקת המידע לפקטות, שליחת הביטים וכו׳, זו עבודה של מערכת ההפעלה:



איור 100: המחשה לאופן פעולת ה-socket בפרוטוקול TCP

כדי לבנות חיבור כזה, הלקוח צריך להתחבר לשרת, מה שאומר שהשרת צריך לרוץ מראש, תמיד, נגיד בלולאה אינסופית.

השרת אם כך, ייצור socket שיחכה לקבלת מידע.

לאחר מכן, הלקוח ייצור socket לשליחת מידע, שבחיבור שלו, הוא ייתן את כתובת ה-IP של השרת, וה-Port הרלוונטי (אנחנו מתחברים לבניין עם ה-IP ולדירה בבניין עם ה-Port).

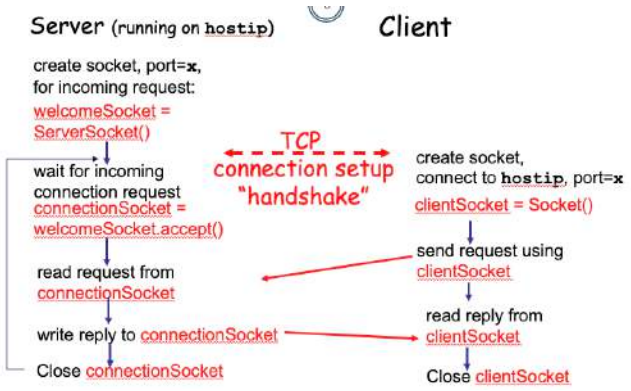
כאשר הלקוח מתחבר לשרת, השרת מקצה עבורו socket חדש, אליו הוא יישלח מידע, הוא שומר גם את כתובת ה-IP של הלקוח. יצירת ה-socket החדש, מאפשרת לשרת לדבר עם כמה לקוחות בו זמנית, נגיד, באמצעות threads - ככה telegram ממומש.

ללקוח ולשרת יש שני ערוצי מידע - intput stream, output stream - מה-socket וקלט המשתמש, ושליחת המידע הלאה.

בסך הכל, כאשר רוצים להקים חיבור TCP, נבצע את השלבים הבאים:

1. השרת:
   (א) ניצור socket שיאזין ב-port = x לבקשות: welcomeSocket = ServerSocket ().
   (ב) נחכה לבקשות נכנסות: connctionSoect = WelcomeSocket.accept ().
   (ג) נקרא את המידע מ-connectionSocket.
   (ד) נשלח תגובה דרך connectionSocket.
   (ה) נסגור את connectionSocket ונחזור לשלב ההמתנה לבקשות נכנסות.
2. הלקוח:
   (א) ניצור socket ונתחבר ל-host_IP, port = x: clientSocket = Socket ().
   (ב) נשלח את המידע דרך clientSocket.
   (ג) נקרא את התגובה של השרת דרך clientSocket.
   (ד) נסגור את clientSocket.

חיבור ה-TCP נכנס בשלב ההתחברות:



איור 101 : התחברות לקוח ושרת ב-TCP

21.2 Sockets בפרוטוקול UDP

כשאנו משתמשים ב-Socket בפרוטוקול UDP, אנו מצהירים שאין חיבור בין הלקוח לשרת, אין "לחיצת ידיים".

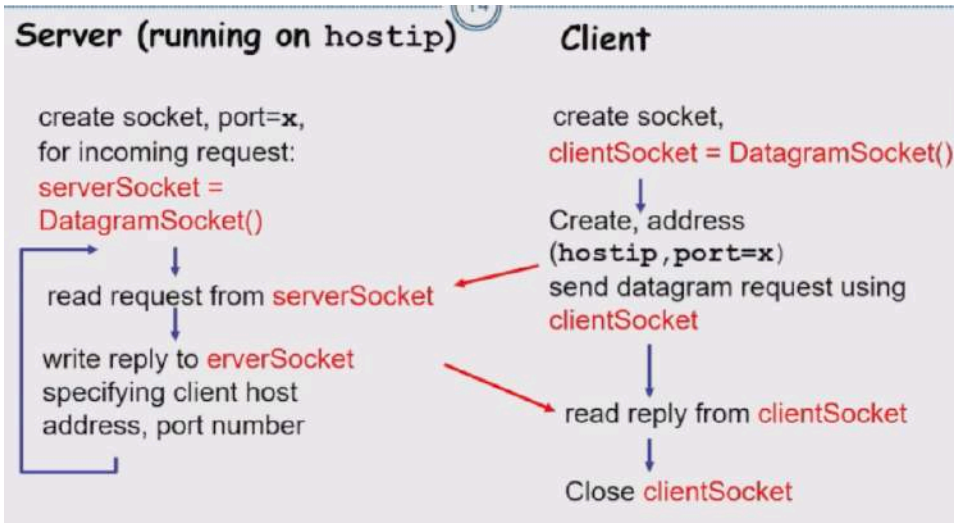
הלקוח מספק באופן מפורש את ה-IP, Port בעת שליחת ההודעה ליעד, עבור כל פקטה, והיא בתקווה תגיע ליעד. השרת מוכרח להסיק את כתובת ה-IP, Port של ההודעה שהגיעה.

המידע עלול להגיע בסדר לא נכון, או לא להגיע כלל, או פגום.

השימוש בו מתבצע באופן הבא:

1. השרת:
   (א) ניצור socket שיאזין ב-port = x לבקשות: serverSocket = DatagramSocket ().
   (ב) נקרא את המידע מ-serverSocket.
   (ג) נשלח תגובה דרך serverSocket.
   (ד) נחזור לשלב ההמתנה לבקשות נכנסות.
2. הלקוח:
   (א) ניצור socket: clientSocket = DatagramSocket ().
   (ב) נשלח את המידע דרך clientSocket (hostip, port = x).
   (ג) נקרא את התגובה של השרת דרך clientSocket.
   (ד) נסגור את clientSocket.

אין לנו חיבור, אלא רק בדיקה אם מישהו שלח מידע:



איור 102 : התחברות לקוח ושרת ב-UDP

21.3 תכנות Sockets

כאשר אנו מייצגים כתובת, אנו משתמשים במבנה:

```
struct sockaddr {
        unsigned short sa_family;
        char sa_data[14];
};
// sa_family is the type of the address, which is in our case an IP address
// to specify that, we pass the value AF_INET to sa_family
// sa_data contains the destination address, and the port for the socket
```

קיימים גם דגלי sa_family שונים, למשל AF_UNIX מציין שהכתובת היא נתיב לתכנית שנמצאת במחשב - שימושי כאשר התקשורת היא בתוך אותו המחשב.

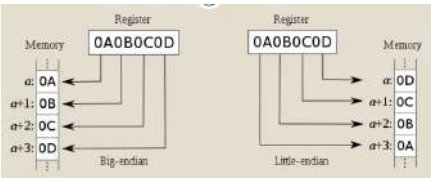
קיים struct נוסף, באותו הגודל, שהוא מיועד יותר לתקשורת דרך האינטרנט. הוא דורש פחות מקום, ולכן המידע הנותר מלא באפסים. הסיבה ששומרים את האפסים, הוא על מנת לאפשר המרה בין המבנה החדש למבנה הקודם, ולהפך:

```
struct sockaddr_in {
        short sin_family; //  should be AF_INET
        unsigned int sin_port; // port
        struct in_adder sin_addr; // IP address
        unsigned char sin_zero[8]; // not needed in a network address
};

struct in_addr{
        uint32_t s_addr;
};

// sin_zero can be set to 0 with memset
```

ה-sin_port, sin_addr צריכים להיכתב בסדר הבתים של הרשת! מה הכוונה?

21.3.1 Big/Little endian

קיימות שתי דרכים לכתוב מידע לזכרון - מבית ה-MSB או מבית ה-LSB:



איור 103: בשיטת ה-little endian אנו כותבים בית "נמוך יותר" לכתובת נמוכה יותר.
בשיטת ה-big endian אנו כותבים בית "גבוה יותר" לכתובת נמוכה יותר. הרשת משתמשת ב-big endian.

ההצדקה לשימוש בשיטות שונות לא מאוד משכנעת.

בכל מקרה, עלינו להמיר את סדר הבתים במחשב לסדר הבתים ברשת. קיימות פונקציות ההמרה:

- htons () - ממיר מה-host לרשת, כאשר הקלט הוא short.
- htonl () - ממיר מה-host לרשת, כאשר הקלט הוא long.

- ntohs () - מהרשת ל-host, כאשר הקלט הוא short.
- ntohl () - מהרשת ל-host, כאשר הקלט הוא long.

נראה דוגמא לשימוש בהן:

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
int main(){
        struct sockaddr_in my_addr;
        my_addr.sin_family = AF_INET;
        my_addr.sin_port = htons(3490);

        // inet_aton convets an IP V4 numbers-and-dots notation into binary
        // non zero on sucess, 0 on failure
        inet_aton("10.12.119.57", &(my_addr.sin_addr));
        memset(&(my_addr.sin_zero, '\0', 8);
        return 0;
}
```

פונקציה שימושית נוספת שמקבלת את המידע מיישות שהתחברה:

```
int getperrname(int sockfd, struct sockaddr *addr, int *addrlen);

/**
* get the address of the other end of a connected stream socket
*    socketfd - the fd of the connected stream socket
*       addr is a spointer to a struct socketaddr that will hold the information about
   the other size of the connection
*       addrlen indicates on the addr's length. Should be initilized to sizeof(struct
   sockaddr). if the value is not big enough, getperrname increases this value
*/
```

21.4 Domain Name Service (DNS)

כאשר אנו מחפשים משאב ברשת, אנחנו פונים לכתובת טקסטואלית. ה-DNS הוא פרוטוקול להמרת הכותבת לכתובת IP.

הפונקציות הבאות נותנות את שם ה-HOST או את כתובת ה-IP:

```
#include <netdb.h>
gethostname(); // returns the name of the computer that your program is running on
struct hostnet*
gethostbyname(const char *name);
// get the name of the host, and returns the ip in hostnet as a pointer, NULL on error

struct hostnet {
        // official name of the host
        char *h_name;
        // alternate names
        char **h_aliases;
        // usually AF_INET
        int h_addrtype;
        // length of each address
```

```
        int h_length;
        // network addresses for the host in N.B.O
        char **h_addr_list;
};
#define h_addr h_addr_list[0];
```

נראה דוגמת הרצה:

```
int main(int argc, char *argv[]){
        strcut hostnet *h;
        if (argc != 2){
                fprintf(stderr, "usage: getip address\n");
                exit(1);
        }
        if ((h=gethostbyname(argv[1])) == NULL){
                fprintf(stderr, "gethostbyname ");
                exit(1);
        }
        fprintf("Host name: %s\n", h->h_name);
        printf("IP address: %s\n", inet_ntoa(*((struct in_addr *)h->h_addr));
        return 0;
}
```

כאשר אנחנו משתמשים ב-Domain של עצמנו נשתמש ב-127.0.0.1.

21.5 תכנות Sockets

21.5.1 צד השרת

השלב הראשון ביצירת השרת הוא יצירת ה-socket.

שלב יצירת ה-socket מקביל ליצירת מכשיר טלפון סלולרי, הדרך לעשות זאת הוא באמצעות systemcall המכונה socket ():

```
int socket(int domain, int type, int protocol);
```

type יכול להיות:

- SOCK_STREAM - פרוטוקול TCP.
- SOCK_DGRAM - פרוטוקול UDP.

protocol יכול להיות "0" כדי שיבחר פרוטוקול דיפולטיבי.

השלב השני הוא חיבור ה-socket לכתובת IP כלשהי. כלומר, אם נקביל זאת לתקשורת סלולרית, זה השלב בו יש לנו מספר טלפון אליו אפשר להתקשר. נבצע זאת עם ה-systemcall הייחודי bind:

```
int bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
```

השלב השלישי הוא להקשיב ללקוחות. כלומר, אם שוב נקבל לטלפון, מדובר בשלב ההמתנה לשיחה. נבצע זאת עם ה-systemcall המכונה listen:

```
int listen(int sockfd, int backlog);
```

דוגמה. נראה דוגמת הרצה לשלושת השלבים:

```
int establish(unsigned short portnum){
        char myname[MAXHOSTNAME+1]'
        int s;
        strcut sockaddr_in sa;
        struct hostent *hp;
        // hostnet initlization
        gethostname(myname, MAXHOSTNAME);
        hp = gethostbyname(myname);
        if (hp == NULL){
                return -1;
        }
        // sockaddr_in initlization (set 0 and then fill)
        memset(&sa, 0, sizeof(struct sockaddr_in);
        sa.sin_family = hp->h_addrtype;

        // connection establishment
        /* create socket */
        if ((s=socket(AF_INET, SOCK_STREAM, 0)) < 0){
                return -1;
        }

        if (bind(s, (struct sockaddr*)&sa, sizeof(struct sockaddr_in)) < 0){
                close(s);
                return -1;
        }

        /* listen */
        listen(s, 3); /* max # of queued connects*/
        return s;
}
```

השלב הרביעי הוא לקבל חיבורים, כלומר, עבור הטלפון שלנו, זה יהיה שלב המענה לשיחה. נעשה זאת באמצעות ה-systemcall הייחודי accept. כלומר, השרת יהיה בלולאה אינסופית עד שמישהו ירצה להתחבר אליו. הפונקציה מחזירה socket חדש שמחובר למי שיזם את חיבור (היישות שהתקשרה אלינו):

```
int accept(int sockfd, struct sockaddr *cli_addr, socklen_t *cli_addrlen);
```

נמשיך עם הדוגמא הקודמת:

```
int get_connection(int s){
        // s was craeted by establish
        int t; /*socket of connection*/

        if ((t=accept(s, NULL, NULL)) < 0){
                return -1;
        }

}
```

21.5.2 צד הלקוח

עלינו ליצור את ה-socket כמו בצד השרת.

לאחר מכן עלינו ספק כתובת להתחברות ולהשתמש ב-systemcall הייחודי connect:

```
int connect(int sockfd, const struct sockaddr *serv_addr, socklen_t addrlen);
```

דוגמה. נראה דוגמת הרצה:

```
int call_socket(char *hostname, unsigned short purtnum) {
        strcut sockaddr_in sa;
        struct hostent *hp;
        int s;

        // hostent initlization
        if ((hp = gethostbyname(hostname))== NULL){
                return -1;
        }

        memset(&sa, 0, sizeof(sa));
        memcpy((char *)&sa.sin_addr, hp->h_addr, hp->h_length);
        sa.sin_family = hp->h_addrtype;
        sa.sin_port = htons((u_short)portnum);


        if ((s = socket(hp->h_addrtype, SOCK_STREAM, 0)) < 0) {
                return -1;
        }
        if (connect(s, (strcut sockaddr *)&sa, sizeof(sa)) < 0 ){
                close(s);
                retrun -1;
        }
}
```

21.5.3 שליחה וקבלת מידע

לאחר שהצלחנו ליצור חיבור בין השרת לבין הלקוח, נרצה לשלוח ולקבל מידע ביניהם.

היות שכל socket מיוצג על ידי "file descriptor", נכתוב ונקרא מידע באמצעות read, write. יחד עם זאת, מספר הבתים שביקשנו לקבל, לא בהכרח הגיעו, לכן עלינו להמשיך לקרוא להן עד שקיבלנו הכל.

דוגמה. קריאת מידע:

```
int read_data(int s, char *buf, int n){
        int bcount; /* countes bytes read*/
        int br;
        bcound = 0; br = 0;

        while (bcount < n) { /* loop until full buffr*/
                br = read(s, buf, n-bcount); // amount of bytes read
                if (br > 0) {
                                bcount += br;
                                buf += br; // advance buffer
                }
```

```
                if (br < 1){
                        return -1;
                }
        }
}
```

21.5.4 שרת מרובה לקוחות ו-select()

השרת מקבל הרבה בקשות להתחברות, ולכן צריך לתת מענה לכמה לקוחות במקביל. כלומר accept תחזיר file descriptors שונים, והוא יצטרך לקרוא ולכתוב לכל אחד מהם בנפרד.

כדי לעשות זאת באופן יעיל, אפשר להשתמש ב-thread לכל לקוח וליצור threads pool.

אבל, דרך אחרת היא להשתמש בפונקציה select:

```
int select(int nfds, fd_set *read-fds, fd_set *write-fds, fd_set *except-fds, struct
    timeval *timeout);
/**
        * nfds -  the maximal fd number to check, plus 1
        * readfds - specifies the file descriptors to be checked for being ready to read
        * writefds - specifies the file descriptors to be checked for being ready to
            write
        * execptfds - specifies the file descriptors to be checked for error conditions
            pending
        * timeout - conrotls how long the select() function shall take before timing out
*/
```

fd_set מייצג קבוצה של descriptors ועליה פונקציות ייעודיות:

- FD_ZERO (fd_set * fdset) - מאתחלת את ה-fd לאפס, עבור כל fd בקבוצה.
- FD_CLR (int fd, fd_set * fdset) - מנקה את הביט עבור fd בקבוצה. הוצאה של fd מהקבוצה.
- FD_SET (int fd, fd_set * fdset) - מגדירה את הביט עבור fd בקבוצה. זה שקול להוספה של fd לקבוצה.
- FD_ISSET (int fd, fd_set * fdset) - מחזירה ערך שונה מאפס אם הביט של fd הוגדר בקבוצה, ואפס אחרת. שקול לפעולה $fd \in fdset$.

נראה דוגמא לשימוש ב-select כפסאודו קוד:

```
MAX_CLIENT = 30;
fd_set clientsfds;
fd_set readfds;

FD_ZERO(&clientsfds); // init clients fds
FD_SET(serverSockfd, &clientsfds); // add server socket to clientsfds
FD_SET(STDIN_FILENO, &clientsfds); // add fd of stdin to clientsfds

while (stullRunning) {
                readfds = clientsfds;

        if (select(MAX_CLIENTS+1, &readfds, NULL, NULL, NULL) < 0){
                terminateServer();
                return -1;
        }
        // check if socket is ready for read, that is, there is a new client we need to
            connect
```

```
        if (FD_ISSET(serverSockfd, &readfds)) {
                //will also add the client to the clientsfds
                connectNewClient();
        }
        // check if stdin is ready, that is, we got data from stdin
        if (FD_ISSET(STDIN_FILENO, &readfds)) {
                serverStdInput();
        }
        else {
                //will check each client  if 'its in readfds
                //and then receive a message from him
                handleClientRequest();
        }
}
```

21.5.5 סיכום

צד השרת משתמש בפונקציות:

1. socket ().
2. bind ().
3. listen ().
4. accept ().
5. read () /write ().

צד הלקוח משתמש בפונקציות:

1. socket ().
2. connect ().
3. read () /write ().

הערה. שני הצדדים משתמשים בפונקציות הממירות כתובות לסדר הבתים של הרשת, ואתחול מבנים פנימיים.

22 תרגול 12 - סיכום

חזרנו על נושאי הקורס.

הערה. fork גורם לתהליך הילד לרוץ מאותה נקודה שהאב עצר בה. ערך החזרה של fork עבור אבא הוא ה-child pid, ועבור הילד הוא 0, אלא אם יש קריאה חדשה.

22.1 תקשורת בין תהליכים

תהליכים יכולים לתקשר באמצעות סיגנלים ו-wait, זכרון משותף, קבצים, sockets ו-pipes.

הפונקציה pipe יוצרת ערוץ תקשורת עבור תהליכים:

```
err = pipe(int fd[2]);
```

שמה שני file descriptors חדשים בתוך fd [1] , fd [0], שמייצגים שני צדדים בערוץ התקשורת, עבור קריאה וכתיבה. הערכים שלהם יהיו הערכים המינימלים, כלומר הראשונים שזמינים בעת ביצוע ה-pipe. קריאהמ-fd [0] היא קריאה מהמידע השמור ב-fd [1], ולהפך. כלומר כתיבה לקובץ אחד מאפשרת לקרוא את המידע באמצעות הקובץ השני.

הפונקציות dup, dup2 כמעט זהות:

```
fd_new = dup(int fd);
```

משכפלת את ה-fd הפתוח ושמה אותו בתוך fd_new, כלומר שניהם מצביעים על אותה **פתיחה** של קובץ, כלומר התקדמות באחד היא התקדמות גם בשני.

```
int dup2(int oldfd, int newfd);
```

עושה אותו דבר רק שה-fd החדש הוא newfd. אם הוא כבר תפוס, אז היא סוגרת את הקובץ שכבר פתוח ותפתח במקומו את הקובץ שביקשנו. מה שאומר שאפשר לדרוס את ה-fd של STDOUT.

נראה דוגמא לשימוש בהם:

```
#include <unistd.h>
#include <stdlib.h>
#include <sys/wait.h>
int main() {

        int pipefd2[2];
        pipe(pipefd2); // create interprocess comunication channel
        // create child process
        if (fork() == 0) {
                // in child

                dup2(pipefd2[1], STDOUT_FILENO); // copy fd to STDOUT, that is, data is
                    not printed to screen anymore
                close(pipefd2[0]);
                close(pipefd2[1]);
                execl("/bin/ls", "ls", NULL); // data is stored in pipe, not in STDOUT
                exit(EXIT_FAILURE); // only if execl failed
        }
        // create second child, and child of child
        if (fork() == 0) {
                // in second child, and child of child

```

```
                dup2(pipefd2[0], STDIN_FILENO); // copy second fd to STDIN, that is, data is not read from screen
                close(pipefd2[0]);
                close(pipefd2[1]);
                execl("/usr/bin/file", "file", "-f-", NULL); // data is stored in pipe, not STDIN
                exit(EXIT_FAILURE);
        }

        close(pipefd2[0]);
        close(pipefd2[1]);
        wait(NULL);
        wait(NULL);
        return 0;
}
```

היות שהילד דרס את STDOUT, ושם בתוכו את רשימת כל הקבצים, הילד של הילד יריץ את file, אך הוא צריך לקבל פרמטרים מ-STDIN, מה שאומר שהוא ינסה לקרוא ממנו, אבל לקרוא ממנו זה לקרוא את המידע מהצד השני של הערוץ, היינו, לקרוא את המידע שנשמר על ידי תהליך הילד, כלומר, רשימת הקבצים, לכן הפקודה תרוץ על כל רשימת הקבצים. הפלט שלה יישמר בצד השני של הערוץ, שזהו בפועל, STDOUT. קיבלנו בעצם תקשורת בין תהליכים.

22.2 קבצים

22.2.1 lseek

הפונקציה

```
where = lseek(int fd, off_t offset, int whence);
// returns the new offset
```

כאשר:

- $\text{whence} \in \{\text{SEEK\_SET}, \text{SEEK\_CUR}, \text{SEEK\_END}\}$:
  - SEEK_SET: ה-$file$ offset משתנה ל-offset.
  - SEEK_CUR: ה-$file$ offset משתנה ל-$file$ offset + offset.
  - SEEK_END: ה-$file$ offset משתנה לסוף הקובץ פלוס offset.
- אם נקרא $\text{where} = \text{lseek}(\text{fd}, 0, \text{SEEK\_CUR})$ נקבל את ה-offset הנוכחי.

22.2.2 hard link

לקובץ יכולים להיות כמה שמות שונים. הקובץ עצמו לא כולל את השם שלו, שכן זה מידע שנשמר בתוך התיקייה שלו. כל שם של קובץ, כלומר מופע חדש שלו ששמור בתיקייה, ומכיל את כל המידע של הקובץ, נקרא hard link. מה שאומר שהם מכילים את השדות של הקובץ ומצביעים לאותו inode.

ליצירה ומחיקה יש את הפונקציות הבאות:

```
err = int link(const char *oldpath, const char *newpath);
```

יוצר hard link לקובץ קיים ומגדיל את ה-ref_count.

```
err = int unlink(const char *path);
```

מוחק את ה-link לקובץ, ומקטין את ה-ref_count. כאשר אם הוא מתאפס, ואין תהליך שעבורו הקובץ פתוח, הקובץ נמחק מהדיסק, ולא נגיש יותר.

אם לפחות תהליך אחד ניגש לקובץ בעת ביצוע הפעולה, ה-link נמחק לפני שחוזרים מ-unlink, אבל התוכן של הקובץ נמחק רק לאחר שכל המופעים של הקובץ נסגרו.

22.2.3 soft link

link מסוג זה, מכיל נתיב ל-hardlink כלשהו. כלומר התוכן שלו הוא הנתיב.

22.3 שאלות

שאלה כיצד ניתן לממש מתזמן תהליכים ב-User Space?

תשובה נוכל להשתמש בסיגנלים SIGSTOP, SICONT שעוצרים את התהליך וממשיכים בהתאמה. לא ניתן לעצור אותן. כאשר תהליך עוצר, מערכת ההפעלה תעבור לתהליך אחר ותעשה swap לזכרון התהליך הקודם. כאשר SIGCONT נשלח, היא תמפה את הזכרון בחזרה. מעל כולם יהיה תהליך אב שישלח את הסיגנלים לפי אלגוריתם שיבחר.

שאלה במערכת קבצים מסורתית מבוססת inodes כמה גישות נדרשות לכל הפחות כדי לקרוא את הבית ה-8097 בקובץ /home/moishe/mail? הניחו שגודל בלוק הוא 1024 בתים, inode מכיל 6 מצביעים ישירים לבלוקים ועוד 3 מצביעים עקיפים

$$(\text{direct, double indirect, triple indirect})$$

כל מדריך (inode) מאוחסן בבלוק יחיד ובתחילת הפעולה נמצא בזכרון רק ה-inode של שורש מערכת הקבצים. נמקו.

תשובה לכל תיקייה צריך לייבא את ה-inode (גישה אחת), לקרוא את התוכן (זו לא עוד גישה, המידע הועתק לזכרון הזמני), ואז לגשת לדיסק בשביל מידע (נגיד לבלוק הראשון). מה שאומר שכל תיקייה דורשת שתי גישות. היות שהתיקייה הראשונה מכילה כבר את ה-inode, כלומר לא צריך להביא אותו, יש לנו $2\cdot 3-1=5$ גישות. כשהגענו לטפל בקובץ, ייבאנו את ה-inode שלו - זה עוד גישה, כלומר 6 גישות. נבחין כי $8 > \frac{8097}{1024} > 7$ מה שאומר שהבית הוא בבלוק השמיני. הבלוק השמיני הוא direct, ולכן דרושה גישה אחת לבלוק ה-direct, ועוד גישה לבלוק עצמו, כלומר עוד 2 גישות. קיבלנו סך הכל 8 גישות.

שאלה להל"ן פתרון לבעיית הקטע הקריטי לשני תהליכים (0 שקול ל-False ו-1 שקול ל-True):

```
// at process $i \in {0,1}$
shared boolean flag[2] = {false};
shared int turn = 0;
do
{
        turn  = 1-i;
        flag[i] = false;
        while !(flag[i-1] && turn==1-i);
        // critical section
        flag[i] = true;
        // remainder section
} while (true);
```

האם המימוש עשוי לגרום ל-deadlock? נמק.

תשובה האלגוריתם מזכיר קצת את האלגוריתם של פיטרסון. כל תהליך מצהיר שהוא לא בתור, מה שאומר שהתהליך השני יקבל True בתנאי הלולאה, ויחכה. אך זה נכון באופן סימטרי לשני התהליכים ולכן שניהם יחכו לנצח. על כן קיבלנו deadlock. למעשה, לא משנה מה, תמיד יתקבל שם deadlock.

שאלה נשאל, האם יתכן deadlock אם נוריד את השורה flag [i] = false ונאתחל shread boolean flag [2] = {true}?

תשובה לא יתכן. נבחין כי הדבר היחיד שמשנה עכשיו הוא turn, שכן flag תמיד true. לא משנה, מה תמיד turn הוא 1 או 0 ולא שניהם, ולכן בדיוק אחד ייצא מהלולאה. כלומר בכל מצב, רק תהליך אחד יכול לחכות בלולאה.

שאלה נניח שמבצעים את השינויים מהסעיף הקודם. האם תכונת המניעה ההדדית מתקיימת? הוכח תשובתך.

תשובה לא מתקיימת. נניח שתהליך אחד נכנס לקטע הקוד הקריטי, זה אומר שהוא יצא מהלולאה כאשר $turn == 1 - i$. התהליך השני מגיע, משנה אותו גם, ונכנס לקטע הקוד באותו האופן.

22.4 מבנה המבחן

- 4 שאלות גדולות עם סעיפים שלא קשורים אחד לשני.
  - ◁ השאלה הראשונה עם סעיפים בלתי תלויים.
  - ◁ שאר השאלות עם סעיפים שיתכן שהם קשורים.
  - ◁ אין בחירה.
- המבחן הוא 3 שעות.
- כל שנה יש שאלה על סנכרון - האם יהיה deadlock? יש מניעה הדדית? וכו'.
- כל שנה יש שאלה על זכרון.
- השאלה האחרונה היא נושא מתחלף - inode/syncrhonization וכו'.
- המבחן ממחושב - נקבל מחברות לטיוטא.

מומלץ לפתור מבחני עבר, באופן עצמאי, להשוות תשובות. ללמוד ביחד.