ב"ה נעשה ונצליח

OS מערכות הפעלה

מרצה: דוד חי

סיכמה: עדי במברגר-אדרי

לתוכן העניינים

כ"ו אדר ה'תש"פ, הרצאה 1, דוד

מטרת הקורס היא ללמד אותנו מה זה מערכת הפעלה ורכיבים בה. נדבר על איך צריך לבנות מערכת מחשב ואיך לארגן אותה.

מהי מערכת הפעלה? מהי מערכת מחשב? מקובל להסתכל עליה כמורכבת מ3 חלקים:

1. יוזר- משתמש
2. אפליקציות
3. חומרה- שהדבר המרכזי בה זה הCPU, המעבד, אבל יש לה גם גישה למערכות איחסון. וגם לזיכרון- הRAM של המחשב, לא דיסק.

מערכת ההפעלה היא המערכת שדרכה האפליקציות יכולות לדבר ולהשתמש בצורה יעילה עם החומרה. נותנת את התשתית לזה. היא מנהלת את המקביליות של התוכניות. מבצעת העברת הודעות דו כיוונית- מהאפליקציה וחומרה או מהחומרה לאפליקציה מסויימת.

היא לא מבצעת דברים בשביל עצמה אלא רק משרתת אותם.

היא תמיד רצה איפשהו. כמעט ואי אפשר להריץ אפליקציות בלעדיה- אפליקציה בקושי יכולה לגשת ישירות למעבד אלא רק דרך מערכת ההפעלה.

מה המטרות שלה? הופכת דברים ליותר שמישים. היא גם גורמת לזה שדברים יהיו בטוחים, ויוכלו לרוץ במקביל בלי זליגת נתונים מצד לצד.

איך היו חיינו בלי OS? הCPU לא היה יודע מאיפה להתחיל לקרוא את הקוד, היינו צריכים לכתוב במקום מסויים מוגדר מראש בזיכרון. היינו צריכים לקרוא ישירות מהמקלדת או מהעכבר. היינו צריכים להגיד לחומרה בדיוק מה לעשות כדי לקרוא זיכרון ממקום מסויים. ככה אפשר די בקלות לדרוס מישהו אחר או לפרמט את המחשב. מקביליות של כמה תוכנות לא תהיה כי אין מי שינהל את זה.

OS נותנות אבסטרקציה, לדוגמא קובץ- שבו נוכל לדבר על איחסון של מידע בדיסק. זה ישאר תקף גם בשינוי טכנולוגיה.

סקירה היסטורית מהדור השני של המחשבים קיבלנו את הבסיס של מערכת ההפעלה. לקחנו את הbatching. כבר היו דברים במקביל בו. לבסיס של זה קוראים SPOOLing- הדפסה במקביל לחישוב, חישוב במקביל לקריאה. הדבר המרכזי זה שיש את הדברים שסיימתי, והרכיב הבא קורה מהדברים האלו. ככה אני מקטינה את הזמן שהמעבד עובד. זה בעצם תור

מונחים: איך אנחנו בודקים יעילות של מערכת הפעלה

- Latency זמן השהייה- הזמן שלוקח לבצע מטלה. נמדד בשניות.
- Throughput תפוקה- הקצב שבו מצליחים לעשות עבודה מסויימת. נמדד בשניות/בתים. זה בעצם נרמול
- Utilization ניצולת- אחוז הזמן שהCPU עושה מטלות מועילות. מריץ שורות קוד שלנו. נמדד בשבר- מספר בין 0 ל-1.

- CPU Usage- אחוז הזמן שהCPU עושה הוראות בכלל (גם דברים לא יעילים), תמיד גדול מהניצולת. נמדד בשבר. לפעמים לא נוכל למדוד את הניצולת ונסתפק ביוזג'.
- Overhead תקורה- כמות הדברים העקיפים שהמחשב מבצע כדי לבצע את הפעולות. בלי זה אי אפשר להשיג את המטרה של לחשב את המידע. יכולה להמדד בזמן, בזיכרון, ברוחב פס של תקשורת ובכל דבר אחר שאנחנו נדרשים לעשות. לא שבר.

בשנות הס60 נוצרו מחשבים שיכולים לטפל במספר תוכניות במקביל שמשתמש ב:

- Multiprogramming- מבוצע אחד אחרי השני, עוברים בין אחד לשני כשאחת התוכניות מסתיימת או כשהיא מוותרת על הקדימות שלה (מבצעת פקודת I\O - למשל כשהיא ניגשת למקלדת, או בכלל להתקן חיצוני שלוקח זמן ולכן אפשר לעשות במקביל חישובים אחרים).
- Time sharing- מערכת ההפעלה יכולה בעצמה להוציא תוכנית החוצה להחזיר אותה אחר כך, התוכנית יכולה גם להצהיר על זה אבל מערכת ההפעלה יכולה גם לבחור לעשות את זה לבד.

  התקורה גדולה יותר כאן, כי למערכת ההפעלה יש גם החלפות יזומות ויש יותר החלפות. זמן ההשהיה יהיה כאן יותר טוב למטלות הבאות בתור.

מספר התוכניות שרצות בכל רגע נתון זה כמספר המעבדים. מעבד מריץ פקודה אחת.

הנחות להרצה במקביל:

- קיים CPU, ולכן לא להשתמש בו יהיה בזבוז
- CPU תמיד מהיר יותר מס/i
- הזיכרון מספיק גדול כדי שנוכל לכתוב בתוכו את כל התוכניות (מדובר על זיכרון המחשב- הRAM)
- DMA- מאפשר ברמת החומרה לכל ההתקנים (דיסק/ מקלדת/ עכבר) לדבר ישירות עם הזיכרון (היום כבר ניתן ממחשב אחד לכתוב לזיכרון של מחשב אחר) כי אחרת לא נוכל להריץ משהו אחר במקביל, כי נצטרך לעבור דרך הCPU
- יש יותר מפעולה אחת לעשות בו זמנית

דרישות ממערכת ההפעלה:

- שמערכת ההפעלה תדע לתקשר עם הס/i
- ניהול זיכרון בין המשימות שרצות במקביל, בלי דריסה של מישהו אחר (שאם הוא מנסה תיזרק שגיאה)
- משהו ששולט על הריצה של התוכניות לתוך והחוצה מהCPU
- משהו ששולט על הפעולות של הס/i האיטיות יותר

לתהליכים יש כמה מצבים שהם יכולים להיות בהם ביחס לCPU: רצים, מחכים, מוכנים שיגיע תורם.

מרכיבים מרכזיים של OS היום שנדבר עליהם בקורס:

- ביצוע פעולות
- ניהול תהליכים במקביל

- ניהול הזיכרון
- תקשורת מול התקני I/O, מה זה דרייברים ולמה הם נחוצים
- מערכות קבצים
- שימוש בshell, GUI, command line
- תקשורת בין מחשבים שונים ובין משימות שונות
- אבטחה בין תהליכים, תיקון טעויות, התמודדות על בעיות

דיברנו עד עכשיו על הדור השלישי והרביעי. בדור החמישי לרוב יש כמה מעבדים (אנחנו נדבר על מעבד יחיד), ווירטואליזציה. היום משתמשים בOS גם בענן, ובפלאפונים.

הגיעו לעצירה מבחינת השדרוג של כל מיני דברים ככה שהאופציה הטובה הייתה לעבור לכמה מעבדים או לכמה ליבות. הבעיה היא שיש חלקים משותפים ביניהם ובאלה שלא.
Virtualization - כאילו מפצל את המחשב ככה שהוא חושב שהוא חלק בפני עצמו עם חומרה (וירטואלית) משלו, ואז הוירטואליזיישן ממיר את הפעולות שהוא שולח לחומרה לפעולת על החומרה האמיתית.
זה טוב בשביל להסתכל על המצב של מערכת ההפעלה (נוכל להעתיק את המידע ולהריץ במקום אחר), להעביר VM ממקום למקום, הוספת והסרת VM לפי הדרישות המשתנות (שפועלות בצורה כמעט לגמרי מבודדת). החסרון הוא שיש עוד שכבה שעושה פעולות עקיפות.
הענן נותן לנו לחסוך בחומרה במחשב שלי ולהשתמש במחשב מרוחק בצורה יעילה. נותן לשתף משאבים.

יש הרבה הבדלים בין מחשב לבין פלאפון סלולרי (יש טבלה במצגת), ויש להם קריטריונים אחרים.
לזכור: Kilo=10^3, Mega=10^6, Giga=10^9, Tera=10^{12}

ב"ו אדר ה'תש"פ, תרגול 1, יובל

להתחיל תרגילים מוקדם.

חזרה על נאנד.

היררכיית הזיכרון מאפשרת לנו לעשות טריידאוף בין המידע שאני רוצה לשמור לבין מהירות הזמן שאני רוצה שדברים ירוצו.
וירטואליזציה- היא הפעלת תוכנה באופן בלתי תלוי בתשתיות המחשב, באמצעות דימוי של תשתיות אמיתיות (ויקיפדיה). יש כמה אפשרויות:

- VM- שכבה נוספת שרצה על מערכת ההפעלה שעליה אפשר להריץ מערכת הפעלה נוספת. הבעיה היא שהשכבה הזו יוצרת המון אובהד, כי היא מקצה מחדש זיכרון עבור כל דבר.
  מפרדיה ברמת מערכת ההפעלה.
- הפתרון לזה זה קונטיינרים- שהם מדמים מערכת הפעלה, אבל אין מערכת הפעלה נוספת באמצע בעצם.
  מפרידה ברמת האפליקציה.

בתרגיל 1 נראה שיש משמעות לדברים האלו, עלויות של קריאה לפונקציה רגילה וקריאה לפונקציה של מערכת ההפעלה. בנוסף יש משמעות לעל מה נריץ (המחשב שלנו, על VM או בתוך קונטיינר).

system call זה משהו שמערכת ההפעלה מריצה. לה יש יותר גישות משלנו והיא בטוחה יותר (בלי באגים).

strace זה כלי שנועד לדבג את הקריאות למערכת השונות שיש.

פקודות בסיסיות:

- man זה הסבר קצר על כל מה שיש בשל
  man 2 open כי יש לopen 2 דפים
- cat קוראת מהקובץ לשל
- wc מדפיס כמה פעמים יש לנו שורות, מילים ובתים בקובץ מסויים

ולגרינד

דוגמא 4 (עד 7): כדי להדפיס מיד את הבאפר שמדפיס למסך: לכתוב בסוף \n, או להשתמש בfflush או לעשות setbuf(stdout, NULL).

דוגמא 2: הוספת g- בשביל דיבאג, מוסיף מידע לתוך ולגרינד.

דוגמא 3: ולגרינד מבקש להוסיף עוד דגלים כדי להבין מה הבעיה.

דוגמא 9: מוצא שימוש אחרי שיחרור זיכרון.

דוגמא 10: שחרור כפול.

דוגמא 8: שימוש בערך לא מאותחל. אם נרצה למצוא אותו גם בלי הדפסה ובלי ולגרינד, נוסיף דגל של Wall- ואז נוכל לראות את זה כבר בקומפילציה.

דוגמא 1: GDB הוא האבא של הדיבגרים (ולא נח).

AddressSanitizer במקום ולגרינד, של גוגל. קורה בזמן קימפול (לעומת ולגרינד שקורה אחר כך וחוצץ בין התוכנית למערכת ההפעלה). אי אפשר להריץ את שניהם.

ד' ניסן ה'תש"פ, הרצאה 2, דוד

יש אינטרפייס בין האפליקציות למערכת ההפעלה- שנקרא קריאות מערכת, ובין החומרה והסטורג' למערכת ההפעלה- ממערכת ההפעלה זה נקרא Privileged instructions ומהחומרה והאיחסון למערכת ההפעלה נקרא פסיקות Interrupts.

מקובל לחלק את הפעולה של הCPU לשני מצבים:

1. ביצוע קוד של המשתמשים- user mode
   לא יכול לגשת ישירות לחומרה
2. שימוש בהרדוור וביצוע פקודות של מערכת ההפעלה- kernel mode (נקרא גם privileged mode)
   יכול להריץ פקודות אסמבלי

system calls נועד להעביר מהמצב הראשון לשני.

למראית עין הם דומים לפונקציות (כמו בנאנד), אבל למעשה זוהי הרצה של קוד של מערכת ההפעלה לפי תיעדוף שלה, העברת השליטה אליה. כדי לפתוח קובץ (משהו פנימי במערכת ההפעלה) צריך להשתמש בהן לדוגמא. הן בודקות האם הקריאה מותרת. זה תלוי מערכת הפעלה.

ויתור על המעבד (קריאה לsystem call)- trap.
בתוך הCPU יש ביט מיוחד שאומר באיזה מצב אנחנו עובדים כרגע. הוא מוודא שנמצא במצב הנכון, ואם הוא לא- הוא אמור לעלות על זה ולזרוק אקספשיין כלשהו.
ניהול אקספשיינים- קורה גם כן באותו מנגנון של trap כדי לתת למערכת ההפעלה להתמודד עם הבעיה (לדוגמא חלוקה בס). גם כשעושים catch בקוד, עדיין יש קוד של מערכת ההפעלה שרץ בין לבין.

סוגים של system calls:

- טיפול בתהליכים
  יש הרבה תוכניות/ג'ובים שמשתמשים באותו מעבד כדי לעשות את הריצה שלהם, מישהו צריך לנהל את זה- חזרה לתהליך שעצרנו, סיום תהליך
- ניהול קבצים (שמירה, כתיבה, קריאה, מיקום הפוינטר שקורא מהקובץ)
  אבסטרקציה שמערכת ההפעלה נותנת לנו
- ניהול התקנים חיצוניים (מקלדת, עכבר- גם קריאה וגם כתיבה)
- מידע שלא שייך ישירות לאפליקציה הספציפית אלא משותף למערכת
- תקשורת
  בין שתי אפילקציות באותו מחשב, ובין שני מחשבים
  דרך סוקט
- אבטחה
  מי יכול להריץ אפליקציה מסויימת, chmod, מי יכול לקרוא מקובץ מסויים. שג'וב אחד לא יוכל לכתוב למרחב הכתובות של ג'וב אחר

לרוב אנחנו לא עושים קריאה בעצמנו למערכת ההפעלה, אלא משתמים בAPI שעושה אבסטרקציה, נותן להעביר את זה בין מערכות הפעלה, ומאפשר בטיחות- לדוגמא printf לעומת write() של מערכת ההפעלה שמקבלת ביטים. אנחנו לא משתמשים בהם ישירות אבל הם נמצאים ברקע.
system programs הן משהו שמגיע עם התוכניות של מערכת ההפעלה, אבל לא חלק ממנה.
לדוגמא הפקודה echo (שמעתיקה את מה שכתבת אחרי). המימוש שלה הוא 274 שורות קוד בC- כולל בדיקת הפרמטרים, שימוש בפונקציות עד השימוש של פקודת הsystem call. כלומר רובו עדיין עובד בuser mode, ויש מעברים בין המצבים (אפילו כמה פעמים) – למרות שהוא ניתן עם מערכת ההפעלה. הגבול בין מהי אפליקציה ומהו שירות של מערכת ההפעלה הוא אפור ולא תמיד ברור. (לדוגמא- האם אינטרנט הוא חלק ממערכת ההפעלה? היום מקובל שלא, בעבר אקספלורר היה חלק אינטגרלי מוינדוס עד שהוצא בצורה משפטית).

לכן מקובל להסתכל על מערכת ההפעלה כעל מספר שכבות:

- אפליקציות
- חלונות, shell
- ספריות- אינטרפייס של מערכת ההפעלה כדי שאפליקציות יוכלו להשתמש בשירותים שלה- stdio.h,java.io
- אבסטרקציות של מערכת ההפעלה- מערכת הקבצים, CPU שמתבצע על ידי תהליכים, תקשורת בין תהליכים/מחשבים שמבצע על ידי sockets
  לאו דווקא קוד, זה קונספט

- דרייברים- החלק שהכי קרוב לחומרה, חלק ממערכת ההפעלה (נכתבה על ידי מי שכתב את מערכת ההפעלה בשיתוף עם החומרה) שמאפשר אינטרפייס לשימוש בhardware
- חומרה

איך בכלל מחברים התקנים חומרה למערכת? מקובל שיש בס (קו תקשורת), שעליו רץ המידע- וכולם מתחברים אליו. היתרונות זה שהוא אחד לכולם (זול) וגם אפשר לחבר ולנתק מכשירים שונים של קלט ופלט. החסרונות זה שהבס הוא צוואר הבקבוק של המערכת (הוא יכול להעביר מידע בצורה מוגבלת, כמו פקק תנועה, דברים הולכים לאיבוד או משתבשים) - הוא יוצר הגבלות על כמות ההתקנים שאפשר לחבר למערכת ויעבדו כמו שצריך. בנוסף, הכל עובר עליו אותו דבר בלי חשיבות לדחיפות.
במציאות מערכת יותר מורכבת ויש כמה בסים שמיועדים לסוגי תעבורה ספציפיים ומונע התנגשויות בין דברים חשובים.

הדרך שבה device מודיע למערכת ההפעלה שהוא עשה את מה שצריך (שסיים את מה שהיה צריך/ שיש לה עבודה עכשיו) זה בעזרת interrupts-פסיקה, שיכולים לקרות כל הזמן- לא קשור לכלום (גם סתם מישהו יכול פתאום ללחוץ על המקלדת). יש חלק חומרתי שלם בתוך המעבד שנועד לטפל בזה. זה נעשה בעזרת IRQ שזה משהו בחומרה.
ברגע שהCPU מקבל אינטרפט-פסיקה (באותו רגע, יש לזה תיעדוף לפני פקודות רגילות), הוא הולך לטבלה שבה יש הסבר על מה לעשות ברגע שמקבל אינטרפט מסויים. יש שם את הכתובות של הקוד של מערכת ההפעלה שצריך להתבצע עכשיו.
רק קוד של מערכת ההפעלה (של הקרנל) יכול לעשות דיסאייבל לאינטרפט, ולעולם לא של היוזר. אם עשו לו דיסאייבל הוא יקפוץ אחר כך שוב, באחריות הdevice.

שלוש דרכים לתקשר עם הOS, בשם כולל נקראים פסיקות:

- Interrupts (hardware\external interrupts)
- System calls - that cause traps (software interrupts)
- Exceptions (when something wrong happens, also considered software interrupts because it is internal)

מערכת ההפעלה מונעת על ידי הפסיקות האלו.

סוגי חלוקות של פסיקות:

סופטוויר (כתוצאה מביצוע של קוד רגיל. כולל בתוכו סיסטם קאלס ואקספשינס- שהגיע מתוך הקוד) והארדוויר (מגיע מלמטה- מהIO)

- פנימית (מתוך הCPU, סיסטם קאלס ואקספשינס)-מלמעלה, וחיצונית (ממקלדת וכו')- מלמטה
- סינכרונים (יחד עם השעון של המערכת) וא-סינכרונים (מהחומרה, בלי קשר למעגלי שעון כי אין להם שעון משותף)
- לפי חשיבות: ניתן למסך (לקבל מאוחר יותר) ולא ניתן למסך (גם אם באמצע ביצוע של משהו אחר או לא מקבל אינטרפטים ברגע, כן מקבל אותו)

- כל זמן מסויים-קבוע (השעון עצמו של המערכת כדי לעורר את מערכת ההפעלה, אחרת מערכת ההפעלה לא הייתה עובדת בכלל) ומתי שנרצה (כל מה שדיברנו עד עכשיו)

ככל שהקרנל קטן יותר יש פחות תקורה, ושימוש בפחות זיכרון, אבל יש פחות הגנה. זה הרבה פעמים אומר שרוב הOS לא נמצא שם, ואם יש שינוי בה לא נצטרך לקמפל מחדש את הקרנל (שזה קימפול מורכב שיכול לפגוע ביציבות המערכת).
היום מקובל שיהיה קרנל מודולרי מונוליטי. יש גם מיקרו קרנל, שהוא פחות נפוץ- כמה קרנלים קטנים שעובדים יחד. ההבדלים הם הבדלים טכניים.

ז' ניסן ה'תש"פ, תרגול 2

המידע שאיתו הCPU עובד נמצא בתוך הרגיסטרים שאיתם הוא עובד (הוא לא יכול לגשת לבד לזיכרון). כל סוגי ההוראות שהמעבד יכול לבצע בסוף מורכבות מ3 סוגי הוראות:

- ניהול המידע- לקבוע רגיסטר, שמירת מידע בזיכרון והצאתו
- פעולות אריתמטית- פעולות ביטוויז, השוואות, פעולות מתמטיות בסיסיות
- control flow - קפיצה בין מקומות מסוימים בקוד, קפיצה בתנאי כלשהו (תת סוג של ניהול המידע, כי משנים את הרגיסטר של הPC)

כל הוראה כזו מתורגמת למספר בינארי.

כשמקבלים פקודת IO היא צריכה לעבור דרך הmemory והcache כי להגיע לCPU. הגישה אליו יותר איטית, אבל היתרון שלו זה שהוא שומר את המידע גם כשהמחשב כבוי.

ניהול מידע בזיכרון וגישה אליו- מהירות לפי הגישה לדברים ברצף/לא. כי לcache נטען בלוק מהזיכרון ולא תא יחיד.

<u>User mode and Kernel mode</u>

הגדרות:

1. **תוכנית** היא קובץ שניתן להריץ אותו- קובץ executable, בשפת מכונה
2. **תהליך** הוא <u>מופע</u> של התוכנית (אפשר ליצור כמה מופעים של אותה התוכנית, גם "במקביל")
   כל תהליך מקבל אזור משלו בזיכרון, עם מקום למשתנים וכו'. בכל רגע נתון יש רק תהליך אחד שקורה במעבד.
3. **קלט/פלט** ניתן להבין בכמה מובנים- המידע מהמשתמש/מקובץ, הפלט זה מה שמודפס למשתמש למסך/מה שנכתב לקובץ. או התקני החומרה ממש.
4. **kernel** הוא הליבה של מערכת ההפעלה, והוא יכול לעשות כל מה שהוא רוצה (יש לו שליטה מלאה על כל מה שקורה במחשב), כי הוא רץ במצב kernel mode של המעבד ולכן הוא אמין (ויכול לעשות הכל, לעומת untrusted שיש לו פחות הרשאות)

הCPU יכול להיות במצב קרנל או במצב יוזר:

1. בuser mode יש הגבלות מסוימות:
   a. לא יכול להריץ פקודות מסוימות
   b. לא יכול לגשת לכל הזיכרון, אלא רק לזיכרון של עצמו (SIGSEG)

c. לא יכול לגשת ישירות לחומרה

2. בkernel mode אפשר לגשת להכל (מניחים שהיא תוכנית אמינה)

הקרנל עצמו (רכיב של מערכת ההפעלה) הוא זה ששולט על הזרימה של התהליכים, ורץ רק במצב קרנל. אליו נתייחס בתור מערכת ההפעלה מעכשיו (במציאות מתייחסים גם לשאר הדברים שבאים איתו שרצים במצב יוזר).

מטרת מערכת ההפעלה:

- לתת לתהליכים לרוץ (ושהיא תרוץ כמה שפחות זמן)
- לאפשר לתוכניות לגשת לדברים שאליהם הן לא יכולות לגשת (ע"י system call, כלומר בקשה לשירות כלשהו ממנה)

סיסטם קול זה המנגון שמאפשר לתוכניות שרצות ביוזר מוד להריץ דברים שהן לא מורשות אליהן בעזרת הקרנל. זה קורה בעזרת אינטרפייס (API) שמערכת ההפעלה נותנת. זה קורה ע"י שינוי המצב (למצב קרנל מוד- מעבר קשה כי דורש הרשאה), ביצוע הבקשה ושינוי המצב בחזרה (למצב יוזר מוד- מעבר יותר קל).

<u>Interrupts</u>

אחריות התהליכים להודיע למעבד שהם מוכנים, ולא להפך.

interrupt controller הוא רכיב (חומרתי) במעבד ששולט על זה. האינטרפטים הם אסינכרוניים, כי התהליך שקורה עכשיו לא יכול לשלוט על זה ולא מודע לזה שהוא נעצר.

יש שני סוגים:

1. מבחוץ- מהחומרה
   נגרמים על ידי רכיב חומרה כלשהו: עכבר, מקלדת, קריאה מדיסק, טיימר
2. מבפנים- מהתוכנה, לא חלק מהריצה הרגילה של התוכנית. נקרא גם אקספשיין.
   קורה במהלך ביצוע התוכנית הנוכחית: חלוקה באפס, סגמנטיישן פאלט, ביצוע הוראה שאסור לבצע, הוראה לא חוקית (מספר בינארי שמייצג הוראה לא קיימת).
   לרוב יגרום לעצירה של התוכנית, אבל יש אקספשיינים שמערכת ההפעלה יודעת איך לטפל בהם. לפני שמחליטים שהתוכנית קורסת, OS נותנת אפשרות לטפל בפסיקה הזו, אבל אם אין כזה אז היא תגרום לתוכנית לקרוס.

יש אינטרפטים שאפשר להתעלם מהם זמנית, ויש כאלו שלא (non-maskable).

הטיפול בו:

1. קבלת הפסיקה (אין באינטרפט פנימי)
   דרך האינטרפט וקטור הוא יודע במה צריך לטפל
2. שמירת המצב הנוכחי של הCPU
   שמירת הפרוגרם קאונטר לדוגמא, וכל שאר הרגיסטרים
3. העברת השליטה וטיפול
   הCPU מטפל בקוד הזה (במצב קרנל, שינוי הPSW)
4. החזרת המצב למה שהיה
5. החזרת השליטה

trap הוא אינטרפט טוב. הוא סוג של אקספשיין שקורה כחלק מהריצה הרגילה של התוכנית. זו הדרך שבה system calls ממומשים.

תהליך של system call:

1. קריאה לפונקציה מהספרייה הסטנדרטית
2. שמירת הארגומנטים במקום בזיכרון מוגדר מראש
3. פקודת trap- העברת שליטה
4. נקראת הפונקציה שמנהלת אותו trap handler, שנקראת גם gate
5. מעבר בין switch case שבודק מה הtrap הזה וביצוע
6. שמירת המידע (הפלט) במקום מוגדר מראש
7. החזרת השליטה ליוזר
8. קריאת המידע שנשמר במקום המוגדר מראש
9. החזרת המידע למשתמש שקרא לפונקציה

תמיד לבדוק את ערכי ההחזרה של system calls! לוודא שהוא תקין.

לגלות אינטרפייסים של דברים:

- man 1 לפקודות shell
- man 2 לsystem calls
- man 3 לספרייה הסטנדרטית של C

י"ב ניסן ה'תש"פ, הרצאה 3, דוד

מבנה המעבד- כמו בנאנד, הכל בחומרה. הוא קורא וכתוב לזיכרון- הוראות וdata.

הפקודות באסמבלי הן על הרגיסטרים ולא ישירות על מה שרצינו.

כדי לדעת איפה אנחנו בביצוע, נסתכל על **ההקשר שבו התוכנית עובדת**:

- בCPU- ערכי הרגיסטרים:
  - PC
  - שאר הרגיסטרים בCPU
  - עוד רגיסטרים מיוחדים (SP, פוינטר לתחילת ההיפ... לרוב מצביע לתחילת המידע)
- בזיכרון: רק לקרוא ממנו לא משנה את הערך שלו
  - קוד/טקסט, מידע, סטאק, היפ
  - מידע חיצוני של הOS שרלוונטי לתוכנית (משתמש, חשיבות)

<u>מושגים נוספים:</u>

תהליכים processes ביצוע של תוכנית בהקשר מסויים (ישות דינאמית שמייצגת את זה).

מבחינתם הם רצים לבד. מה שחשוב, חוץ מהתוכנית עצמה, זה ההקשר שבה הוא נמצא, ואם הוא ברגע רץ בCPU (הדברים שלו כתובים על הרגיסטר- הPC מצביע עליו) או לא (ההקשר שלו שמור בזיכרון). ההקשר אומר לנו מה צריך לשמור בצד כשמחליפים בין תהליך אחד לאחר.

<u>לא להתבלבל!</u> **תוכנית**- רצף של פקודות שמישהו כתב (המתכון להכנת עוגה). **תהליך**- משהו דינאמי, שעלה לזיכרון ויש לו היפ, סטאק, והקשר. מדובר על הביצוע של התוכנית.

התוכנית היא חלק מהמצב של התהליך. לתוכנית אחת יכולים להיות יותר מתהליך אחד.

מרחב הכתובות של תהליך כתובות וחלקים מהזיכרון שמותר לתהליך מסויים לגשת אליהן. נדבר על איך זה מנוהל לקראת סוף הסמסטר. מוגדר ע"י שני פרמטרים- bound כמה מותר לו לצרוך, base כתובת התחלתית (וכל הזיכרון מוקצה באופן רציף, [בייס, בייס+באונד]). כל פעם שהתהליך

ניסה לגשת למקום אחר המעבד לא נתן לו (נזרקה פסיקה). כשתהליך רוצה לגשת לכתובת מסויימת, הוא בעצם מתכוון לבייס+מה שרצה (נבדק האם המספר הזה הוא לא יותר מהבאונד). הכתובות האלו נשמרות גם ב2 רגיסטרים נוספים.

הביט הראשון אומר באיזה מצב אנחנו sysmode (לפעמים אנחנו בזיכרון של מערכת ההפעלה ומריצים קוד שלו, אבל עדיין במצב יוזר).

לזיכרון של מערכת ההפעלה (שהיא משתמש בו) אנחנו קוראים kernel memory.

לפי הבייס והבאונד הCPU יכול לדעת שהוא מריץ תהליך של מערכת ההפעלה, ובכלל יודע אם הוא מריץ תהליכים שונים.

<u>מה צריך לשמור כדי לעבור בין התהליכים?</u> ההקשר- את ערכי הרגיסטרים. הזיכרון מוקצה רק לי ולכן אף אחד לא ישנה אותו, אז לא צריך לעשות כלום. לדוגמא: צריך לשמור את הPC אבל לא את הקוד, את הSP אבל לא את כל הסטאק.

איפה שומרים את זה? בתוך הזיכרון ששייך למערכת ההפעלה- PCB program control block.

היא מעדכנת את זה רק כשמחליפים תהליך.

מה שומרת על התהליך?

- מצב
- מספר
- ערכי הרגיסטרים, PC וכו'
- הגבלות הזיכרון (בייס ובאונד)
- רשימת הדברים שהOS פותחת בשביל התהליך (קבצים פתוחים ואיפה נמצאים בהם, תקשורת)
- נתונים על עדיפות, המשתמש שזה רץ אצלו, פרמטרים של GUI

לכל תהליך יש כמה מצבים, כשהוא בעיקר עובר בין 3 המרכזיים:

- new
- ready או blocked
- running- רק אחד בכל פעם
- waiting
- terminated- סיים ולכן לא נחזור אליו שוב

המעברים הם בעזרת אינטרפטים- יציאה לIO או יזומה של הOS במצב מולטיפרוגרמינג. מערכת ההפעלה מחליטה מי יקבל את הCPU ולכמה זמן. היא בוחרת מבין מי שנמצא בready מי יעבור להיות במצב running. במולטיפרוגרמינג- מחליטה גם כמה זמן הוא יכול להיות שם (אם הוא לא יצא משם לפני זה).

מנוהל ע"י תורים.

לרוב מחלקים לשתי ישויות נפרדות:

- CPU Scheduler המוח, מחליט מי ירוץ (ולפעמים לכמה זמן)
- dispatcher המבצע, מי שמחליף את המצבים (מחליף חזרה למצב יוזר)- מעתיק את הערכים של מי שנמצא לזיכרון ואת של מי שנכנס לרגיסטרים

fork לפצל תוכנית שלי לשני תהליכים באופן יזום (פקודה בשפה גבוהה לסיסטם קאל)- מה שחוזר לpid ממנו זה 0 לבן, ולא 0 לאב. כך אנחנו יודעים איפה אנחנו נמצאים.

י"ב ניסן ה'תש"פ, תרגול 3

threads and signals

אינטרפט- נוטיפיקציות למערכת ההפעלה שמטופל על ידה. סיגנלים- נוטיפיקציות ממערכת ההפעלה שמיועדות לטיפול ע"י התהליך.

<u>איך זה קורה?</u>

- קלט מהמשתמש (לדוג' קונטרול+C, קונטרול+/ שמסיימים תהליך, קונטרול+Z שמסיים תהליך עד להופעת ההחזרה שלו), או פקודה של kill מהשל, או לפונקציה kill
  - דרך kill שולחים את כל הפקודות, אפשר לבדוק את כל האופציות עם kill –l
  - ניתן לכתוב בעצמנו סיגנלים בין תוכניות SIGUSR1
  - strace עוקב גם אחרי סיגנלים
- שעון מעורר ממערכת ההפעלה לתוכנה, או מפרוסס אחר שרץ (נניח בליבה אחרת)
- אינטרפט שנגרם מפקודה לא חוקית (חילוק בס)

תהליך יכול שיהיה לו משהו שמנהל את הסיגנלים. אבל יש סיגנלים שעליהם אי אפשר לשלוט בתוכנית (KIILL,STOP). משתמשים לזה בפונקציה signal שהיא חד פעמית וקורית בפעם הבאה שזה יקרה (כבר לא משתמשים בה). אם נרצה להתעלם נקרא לו עם SIG_IGN וכדי להחזיר לרגיל נכתוב SIG_DFT.

התעלמות זמנית מסיגנלים עד שנבטל את החסימה- בעזרת sigprocmask שמקבלת פקודה של סיגנלים וחוסמת אותם, או לעדכן את הקבוצה החסומה כרגע להוסיף גם את אלו שקיבלה, או לשחרר את מה שקיבלה.

הפונקציה sigaction היא קבועה, והיא חוסמת סיגנלים בו זמנית במהלכה.

אלו שחסמנו לא נמחקים, הם מחכים עד שנשחרר את החסימה.

מה זה thread? הוא חי בתוך תהליך, ויכולים להיות כמה כאלו באותו תהליך. לכל אחד מהם יש סטאק ומצב של רגיסטרים משלו. ובמשותף את הקוד, הזיכרון (היפ, וגם גישה למשתנים מקומיים), קבצים פתוחים וכו'. (לדוגמא: שמירה בוורד ובמקביל כתיבה לקובץ הפתוח)

יש 2 סוגים של טרדים:

- ברמת היוזר
  - אנחנו מממשים בעצמנו בלי שמערכת ההפעלה מודעת
    - נשמר- PC, SP, מצב הCPU
    - לא נשמר- משתנים גלובליים (הם משותפים), ההיפ, משתנים לוקאליים
  - *מחיר הקונטקס סוייץ הוא פחות יקר* (תקורה נמוכה יותר כי לא מערבים את הOS)
  - הבעיה בזה שמערכת ההפעלה לא יודעת זה שהיא חוסמת את כל התהליך וטרדים אחרים לא יוכלו לרוץ, גם אם רק אחד היה אמור להעצר (קריאה של אחד לIO)
  - איך עושים?
    - sigsetjmp כדי לשמור- איך נדע אם אנחנו אחרי הקריאה לפונקציה או אחרי הביצוע? בעזרת ערך ההחזרה (0 אם הרגע שמרנו, ערך אחר כשנשחזר)
    - siglongjmp כדי לחזור- מקבל את מה ששמרנו בsigsegjmp ומשחזר אותו, ואת ערך ההחזרה הרצוי לsigsetjmp
  - ניתן לשנות את *הסקודואלר שיהיה מתאים ספציפית לתוכנית*
- ברמת הקרנל (שהניהול קורה ע"י הקרנל)

- קונטקס סוויץ יותר יקרה, אבל *טרדים אחרים יוכלו לנצל את המעבד* אם אחד יוצא לIO, ואם יש כמה ליבות אז *כמה יוכלו לנצל אותם במקביל*
- *הסקדואלר עושה בחירות יותר אינטיליגנטיות*

מה עדיף? תלוי מה עושים (הרבה פקודות IO או חישוב מתמטי. אם אין כמה ליבות אלא רוצים למזער תקורה).

כ"ה ניסן ה'תש"פ, הרצאה 4, דוד

כמו שיש יצירה של פרוססים, גם יש process termination:

- נגמרת התוכנית- יציאה נורמלית
- טעות בתוכנית עם קריאה לexit
- תקלה (חילוק בס)- יציאה לא מרצון
- תהליך אחד יכול להרוג תהליך אחר באמצעות signals

IPC – התקשורת בין שני תהליכים (פיצול משימה גדולה בין תהליכים שונים), צריך מאוד להזהר בה כי רוצים שהם לא ידרסו אחד לשני את הזיכרון. מסופק ע"י הOS באמצעות system calls ספציפיים. התקורה מאוד גדולה.

דרכים לעשות את זה:

- להגדיר חלק בזיכרון שיהיה משותף לכמה תהליכים, ואז אחד יכתוב אליו ואחר יקרא ממנו- וכך יוכלו להעביר את האינפורמציה אחד לשני. יחסית לא כבד
- שליחת הודעה מתהליך אחד לשני. מבחינת הOS זה כמו לשלוח ממחשב אחד לשני. פעולה כבדה מאוד

יוצר בעיות של סינכרון בין תהליכים.

threads – גישה "רכה" יותר של process.

1. מסלול יחודי של ביצוע הפקודות של התוכנית.
2. לכל תהליך יש thread of control אחד

רוב האפליקציות צריכות לבצע יותר ממשימה אחת בו זמנית. איך גורמים להן לעבוד ככה?

3. לא לאפשר את זה לתהליך אחד. לא מומלץ..
4. לפצל כל משימה לתהליך. הבעיות:
   - דברים מאוד תלויים אחד בשני, צריכים לשתף ביניהם הרבה מידע (הרבה IPC)
   - הקוד משוכפל בזיכרון המו פעמים
5. לכן נעדיף multi-threaded processes שיש להם יותר חלקים משותפים
   - thread ששייכים לאותו תהליך- לכל אחד יש רגיסטרים וסטאק שונים, אבל משתפים את הקוד, את שאר המידע (משתנים גלובליים, משתנים בהיפ) ואת הקבצים

thread יכול ליצור thread חדש. הם יכולים גם יותר לדרוס אחד את השני. אנחנו מתפשרים כאן יותר על isolation (בידוד בין התהליכים) כדי לקבל יותר גמישות ופחות IPC.

מי מנהל את הthreads? תלוי.

<u>user-level threads</u>:

6. האפליקציה מנהלת את הthreads שלה, ביוזר מוד
7. הקרנל לא מודע לקיום של הthreads

8. המעבר בין טרדים לא דורש הרשאות של קרנל מוד

אבל, הבעיות:

9. אם thread אחד עושה system call, מבחינת הקרנל כל הprocess נחסם (כל היתרון של הרצת משימות במקביל כבר לא יקרה)
10. אם לא מתכנתים כמו שצריך (אי אפשר להוציא בצורה כפויה לדוגמא, כי אין עזרה של הOS, אלא רק אם תהליך מוותר בעצמו על המעבד), thread אחד יכול להשתלט ולקחת את כל הזמן של התהליך

<u>threads שמנוהלים ע"י מערכת הפעלה:</u>

11. מתייחסת לthreads כמעט כמו לתהליכים, אבל עדיין יש להם את החלקים המשותפים
12. בטבלה של התהליכים (עם הPCB- process control block), במקום של התהליך יהיה מצביע לטבלה של הטרדים שרצים עליו (TCB), וכל טרד יכנס למקום הרלוונטי אליו בתורות (ready וכו'). התעדוף יהיה לפי תהליכים ואז לפי טרדים בתהליך.
13. בלוק לthread אחד לא עושה בלוק לאחר

אבל, הבעיות:

14. מעבר בין threads דורש את הקרנל ולכן יקר יותר

לשים לב:

- יש kernel-level threads שמנוהלים ע"י הקרנל אבל רצים ביוזר מוד
- יש kernel threads שמנוהלים ע"י הקרנל ורצים במצב קרנל (threads של הOS)

<u>יתרונות של threads:</u>

15. תהליך קל משקל, יותר קל ליצור אותו (פי 30-100 מהיר יותר)

16. יותר קל להרוג אותו

17. יותר מהיר לעשות החלפה בין טרדים (בערך פי 5 מאשר בין תהליכים)

18. טרדים של אותו תהליך משתפים ביניהם מידע, ולכן יותר קל, זול ונח לשתף ביניהם מידע בלי התערבות של הקרנל

הCPU לא מודע לקיום של אף thread (הOS לא מודע רק לטרדים ברמת היוזר).

במציאות, מריצים כמה CPU במקביל כדי לבצע פעולות (multi processors/multi core). זה ממש כמו עבודה בנפרד, אבל אם התהליכים רוצים לתקשר ביניהם- זה יקר יותר.

hyper threading – hardware level threads אלו דברים שהCPU מודע אליהם, בלי מעורבות של מערכת ההפעלה. לה זה נראה כאילו יש כמה cores למרות שיש רק אחד. לדוגמא במימוש, למערכת ההפעלה יש שני סטים של רגיסטרים. בדר"כ היום לכל core יש שני hyper-threads.

סינכרוניזציה

הcontext switch הן ברמת פקודות אסמבלי.

נדבר על הסינכרון בין תהליכים באמצעות ה"פתקים" שמשאירים.

הבעיה היא שיש כניסה למשאב משותף בלי תיאום בין הישויות שמבצעות את הכניסה למשאב המשותף (כמה טרדים, כמה תהליכים, כמה קורים/פרוססורים). נעשה במקום שהוא קריטי לקוד (פקודה אחת או קטע קוד גדול).
הפתרון הוא שבמהלך ביצוע קטע הקוד הקריטי, רק תהליך אחד מבצע פעולה. נקרא "מניעה הדדית". אבל לא תמיד זה מספיק, וצריך תיאום של דברים אחרים.

כתיבת קוד כניסה ויציאה כדי לוודא את התכונות הבאות, שצריך לקיים כדי שהדברים יעבדו:

1. מניעה הדדית mutual exclusion – לא יכול להיות ששני תהליכים ייכנסו לקטע הקוד ביחד
2. התקדמות progress – אם תהליך רוצה להכנס, אז תהליך מתישהו יכנס
3. מניעת הרעבה starvation freedom – אם תהליך רוצה להכנס, אז הוא מתישהו יכנס
4. כלליות generality – האלגוריתם יעבוד בכל מקרה
5. no blocking in the remainder לא לשכוח לשחרר את המנעול

איך נעשה את זה? ניסיונות הנחה חבויה- קטע הקוד נגמר בזמן סופי

I. לעשות disable interrupts בהתחלה וenable בסוף.
   a. 1,2,3,5 עובד- 4 לא, כי זה לא חוסם את כל הCPU אם יש מולטי-קור.
   b. זה לא משהו שצריך לעשות בתהליכים של היוזר
   c. זה צריך לקרות לזמן קצר (ויכול להיות שקטע הקוד שכתבנו בפנים ארוך)
II. משתנה גלובלי שנועד לשימוש הכניסה והיציאה לא עובד
   a. mutual exclusion מתקיים- הוכחה במצגת
   b. אבל 2 ו5 לא עובדים

כ"ח ניסן ה'תש"פ, תרגול 4

טבלת סיכום על ההבדלים בין תהליכים, kernel level threads וuser level threads- במצגת.
איך מנצלים את הmultiple threads:

1. חלוקה למשימות קטנות
2. במקום לבצע כל משימה אחת אחרי השניה, נשלח אותה לthread וניתן להכל להתבצע בו זמנית

   הבעיות הן:
   a. זה יכול להעמיס על המשאבים והזיכרון של המערכת.
   b. יכולים להיות משאבים שנרצה להגביל את הגישה אליהן
   c. יש overhead מיותר על כל היצירה והמחיקה של הטרדים
3. בגלל זה יש **thread pool**- פעם אחת אנחנו יוצרים כמה, וכל פעם שיש משימה חדשה- אנחנו שולחים אל thread מסוים משם את המשימה, וכשהוא מסיים לבצע אותה הוא חוזר להיות זמין למשימות חדשות.
   a. בעיות- דליפת טרדים: טרד מסויים קורס (אקספשיין) או נתקע (מחכה לפעולת IO שלא מגיעה)- ולכן נגביל אותו לזמן מסויים שאז נעצור אותו
   b. פתרונות: להגביל את הזמן שמחכים לפעולת IO, ולתפוס אקספשנים

אז עדיף pool גדול או קטן? החסרונות של כל אחד

1. גדול:
   a. הזמן של ליצור את כולם (אבל לא נורא, כי קורה רק פעם אחת)
   b. עלולים לגמור כל מיני משאבים בתוכנית
   c. הרבה תקורה של context switch
2. קטן:
   a. לא ננצל את המעבד כמו שצריך- כולם בblocked כרגע, המעבד פנוי ויש משימה אבל אין טרד פנוי
   b. משימות קצרות יתבצעו לפני משימות ארוכות, אבל אם יש מעט טרדים וכולם יהיו תפוסים- משימה קטנה תחכה הרבה זמן

אין לזה פתרון קבוע, אלא תלוי בעומס. אפשר גם לעשות תוכנית שמשנה את הגודל של הpool תוך כדי ריצה. נרצה estimated average blocked time/estimated average service time +1 טרדים. זה לא פתרון תמיד אלא כשהמטרה היא שהתהליכים ימשיכו גם בזמן שמחכים לפעולות קצרות כמו IO.

איך יוצרים טרדים בלינוקס? pthread_create. וכדי לסיים pthread_exit, או חזרה מהפונקציה, או שטרד אחר קורה לpthread_cancel, או שהתוכנית הסתיימה- אחד הטרדים קרא לexit, או פונקציית הmain החזירה ערך.

אנחנו מעוניינים **לסנכרן** בין טרדים. pthread_join מאפשרת לנו לבצע פעולה רק אחרי שטרד מסויים נגמר (שאותו מקבלים כארגומנט).
בגלל שאנחנו אף פעם לא יודעים מה הסדר שבו הדברים יקרו, אנו רוצים לעשות אלגוריתם למניעה הדדית לקטעי קוד קריטיים. אחת השיטות להשתמש בזה היא mutex- כמו מנעול, שרק טרד אחד יכול לנעול אותו כל רגע. כל שאר הטרדים הופכים לblocked. כשהוא מסיים, הוא פותח את המנעול וגם טרדים אחרים יוכלו לבצע את קטע הקוד הזה אחד אחרי השני. בסיום, הורסים את המיוטקס.
צריך להזהר מחסימה הדדית שתגרום לdeadlocks!
מוניטורים-**conditional variables** עוזרים ל:

1. דברים שיקרו אחרי/בתנאי שדברים אחרים קרו (bounded buffer כמו המשימות לטרד פול)
2. מקום שאנחנו לא רוצים ששני טרדים יכתבו בו בו זמנית, או שיתבצעו קריאה וכתיבה בו זמנית, אבל אפשר שכמה יקראו יחד.
3. מחסום- כמה טרדים שמבצעים יחד משהו, אבל זה מחולק לשלבים (כמו merge sort)

אלו דברים שלעשות עם מיוטקס זה לא טוב, כי לדוגמא נעשה busy wait שזה מיותר, נעדיף ללכת לישון ושיעירו אותנו כשמה שאנחנו מחכים לו קרה.
יש לנו 2 פונקציות מרכזיות:

- wait- הולך לישון ומחכה שסיגנל יעיר אותו. מקבלת conditional variable וגם את המיוטקס. הפונקציה חוסמת אותנו ומשחררת את המיוטקס ששלחנו.
- signal- מעיר אותנו, וחוסם את המיוטקס שוב.

הגיוני שמי שיתעורר ראשון, אם כמה ישנים, יהיה לפי סדר הגעה- אבל זה לא מובטח.

semaphores הכללה של מיוטקסים לכמות מסויימת של טרדים שיכולים להיות בקטע קוד קריטי בו זמנית. נרצה לעשות את זה כשיש לנו משאבים מוגבלים. אפשר גם לשתף אותו בין תהליכים שונים (לרוב לא נרצה). גם כאן יש פונקציות של wait וpost. אם היו טרדים שנחסמו כי ניסו להוריד את הכמות מתחת לס, הם עכשיו יתעוררו ויורידו את הכמות.

ב' אייר ה'תש"פ, הרצאה 5, דוד

קשרים בין תכונות ההתקדמות:

- העדר הרעבה ← progress ← no blocking in the reminder
- blocking in the reminder ← אין progress ← אין העדר הרעבה

הוכחת mutual exclusion – הוכחה בשלילה שקיים. progress – אפשר להוכיח שלא מתקיים (חסימה הדדית- מעגל חסימות). no blocking in the reminder – הסר על המשתנה החוסם.

III. שימוש ב2 משתנים- תור ודגל. לא עובד
   a. אין mutual exclusion!
IV. זה עובד- אם הופכים את השורות: אלגוריתם peterson's
   a. מתקיים mutual exclusion – הוכחה: נניח בשלילה ששני הקטעים נמצאים בקטע הקוד הקריטי. נניח בה"כ שטרד 0 הוא הראשון שנכנס. זה אומר שקרה אחד משני הדברים:
      i. flag[1]==false : זה אומר שטרד 1 לא ביצע את השורה הראשונה שלו כשטרד 0 נכנס- בשורה 3 שלו.
      ii. turn==0 :
   b. no blocking in the reminder מתקיים באופן ברור
   c. העדר הרעבה – הוכחה במצגת, וזה מוביח גם שמתקיים progress

חסרונות- עובד רק ל2 טרדים, ומשתמש בbusy wait.

3 תופעות מרכזיות שגורמות לבעיות:

- race condition – עבור תיזמון מסויים, שקורה מדי פעם, נגיע לדד-לוק. לא קורה כל הזמן אלא תלוי scheduler. קשה מאוד לדיבוג.
- atomic instruction – פעולות שלא יכול להיות context switch באמצע, שהן פקודת מעבד יחידה.
- busy-waiting (spinning, busy-looping) – לולאות שלא עושות כלום אלא מחשבות תנאי עד שהוא נשבר.

הכללת האלגוריתם של פיטרסון לn טרדים- כמו תור לרופא. זה עובד בעולם אידיאלי, שבו הפקודות האטומיות מאוד מוגדרות. יכול להיות מצב ששני טרדים יקחו את אותו מספר יחד! כי כדי לקחת מספר צריך לקרוא את המספר ולעדכן למספר הבא (2 פקודות אטומיות לפחות). נשבר mutual exclusion. עשינו שכלולים בעזרת העדפה לקסיקוגרפית, שגם לא עובדת (ם 5 רץ עד קטע הקוד קריטי כש4 עוד בוחר מספר).

לכן נוסיף משתנה שאומר שתהליך מסויים בוחר מספר עכשיו, ונחכה לו. עכשיו זה עובד.

במבחן יכולים לשאול האם 5 התנאים מתקיימים. כאן מתקיים גם FIFO עד רמה מסוימת.

היום המעבדים תומכים בפעולות מסובכות יותר שיהיו אטומיות RMW:

- test&set – בדיקה אם כתובת ריקה, כתיבה של 1 אליה, והחזרת הערך הקודם (0 או 1)
- fetch&add – x=x+1 בצורה אטומית- או הוספה של כל 2 מספרים.
- compare&swap – השוואת ערך תא לערך חדש. אם הערך שונה מהערך הישן, מוחזר שקר. אם הוא שווה, נכתב הערך החדש ומוחזר אמת.

יש אלגוריתם שמשתמש בהם כדי לפתור את הבעיה.

יש בעיות אחרות של קטעים קריטיים, ובעיות אחרות של סנכרון.

<u>פרימיטיב של סינכרוניזציה</u>: ממשק של מערכת ההפעלה, מבנה נתונים כלשהו שנוכל להשתמש בו, שנותן למערכת ההפעלה לדאוג לזה. לכן יהיה הרבה יותר כח. יוריד את הצורך לעשות busy-wait.

טיפוס אחד שנלמד: semaphore – יש לו 2 שדות: ערך, ורשימה של תהליכים. ניתן לאתחל אותו לערך מסויים, להוריד ערך ולהעלות ערך. אם הערך קטן מאפס, התהליך שביקש להוריד את הערך נכנס לרשימה והולך לישון. אם הבקשה הייתה להעלות, מעירים טרד אחד. (אנשים שמחכים בתור למסעדה ומקום שמתפנה). אלו פעולות אטומיות!

כל התכונות מתקיימות, אבל רק אם הרשימה FIFO מתקיים מניעת הרעבה.

לסמפור בינארי (מוגבל לערך 1 וס, לא אכפת כמה אנשים מחכים בתור אלא רק אם מחכים) קוראים מיוטקס.

הרצה של קטע מסויים אחרי קטע אחר- עם סמפור שמאותחל לס.

סמפור לא פותר את כל הבעיות. לדוגמא- העברת כסף בין שני חשבונות בנק.

לכתוב את ההבדלים בין האלגוריתם- של ברס, של פיטרסון.

ב' אייר ה'תש"פ, תרגול 5, יובל יעקובי

מה שנדבר היום רלוונטי גם לתהליכים וגם לתהליכונים.

האלגוריתם של פטרסון מממש את כל התנאים חוץ מgenerality.

עברנו על 3 בעיות.

תרגיל 3- Map reduce- הפרדה של בעיה גדולה לכמה קטנות.

ט' אייר ה'תש"פ, הרצאה 6, דוד

תזכורת- סמפורים. בעיית סינכרון שניתן לפתור אותה בקלות בעזרת סמפורים- בעיית המניעה ההדדית, וביצוע משימה אחת אחרי אחת אחרת.

סמפור הוגן- אם מתקיים גם FIFO.

בעיית העברת כסף בין חשבונות בנק, עבור כמה חשבונות:

- מנעול יחיד להכל- לא יעיל עבור העברות לא קשורות (Am לB, Bm לC, Cm לD- אין סיבה שהראשון יחסום את השלישי)
- מנעול לכל חשבון- המנעולים מסודרים לפי הסדר שהחשבון שמעביר ינעל ראשון. יכול להיות דד-לוק אם ההעברות הן מעגליות (נוסיף העברה Dm לA). הפתרון הוא לסדר לפי אותו סדר בכל פעם- לנעול מההתחלה לסוף ולשחרר מהסוף להתחלה.

בעיית הפילוסופים האוכלים- הם חושבים או אוכלים (זמן סופי) ע"י צ'ופסטיקים- יש להם צורך ב2 כאלה זהים כדי לאכול. מספר המקלות הוא כמספר הפילוסיפים, כל מקל יכול להיות רק אצל פילוסוף אחד. פתרונות:

- סמפור לכל מקל, כל אחד מנסה לתפוס את מה שמשמאלו- יכול להיות דד-לוק אם כולם תופסים את מה שמשמאלם. נוצרת המתנה מעגלית.
- להחליט שאחד מתנהג הפוך- מנסה קודם להרים את הימני שלו, ואז את השמאלי שלו.

שברנו את הסימטריה. בצורה פורמלית- צריך להוכיח progress. נניח בשלילה שאף אחד לא הצליח לאכול, נפריד למקרים.

ארבעה תנאים כדי שיקרה דד-לוק (כולם צריכים להתקיים):

1. מניעה הדדית- קיים לפחות משאב אחד שצריך להנעל
2. hold and wait- יש יותר ממשאב אחד שצריך, ויש מצב שבו נעלתי מצב אחד ואני מחכה לאחר
3. non-preemptive allocation- מי שנעל משאב הוא היחיד שיכול לשחרר אותו
4. circular wait- מעגליות

הצגה של הבעיה בגרף, כדי להראות מעגליות, ברגע נתון של התוכנית. אין דרך לגלות את זה בצורה חד משמעית.

פתרון נוסף, טוב יותר לבעיה:

- חלוקה לטרדים זוגיים ואי זוגיים- זוגיים לוקחים קודם את ימין ואז את שמאל, אי זוגיים להפך. ואז לפחות 2 יוכלו לאכול יחד, ולא רק אחד. בכה יש מקביליות יותר טובה.

זו בעיה מאוד גדולה, כי היא לא קורית תמיד ולכן קשה לעלות על זה. הדרך הטובה ביותר להסתדר עם זה היא למספר את המשאבים, כך שהנעילה קורית מהמספר הנמוך לגבוה (או בכל סדר אחר, חשוב שיהיה באותו סדר תמיד), והשחרור מהגבוה לנמוך (להפך ממש שנעלנו). כך לעולם לא יהיה לנו מצב של מעגל.

לחשוב למה בשאלה השלישית שני התנאים מתקיימים.

בעיית producer-consumer: כמה יצרנים וכמה צרכנים של משימות.

מה קורה אם הגודל של הבאפר חסום? לרוב מסתכלים על באפר ציקלי- שבו אם סיימתי למלא את הסוף, אחזור למלאה מההתחלה (בתקווה שהראשון קרה כבר). במקרה כזה, יש בעיה גם רק עבור יצרן וצרכן אחד. במצב בו שניהם מצביעים על אותו מקום, אנחנו לא יכולים לדעת אם הבאפר ריק או מלא. אפשר להוסיף קאונטר, שיחזיק מידע על כמה איברים יש בבאפר כרגע. הפעולות של הוספה והחזרה חייבות להיות אטומיות- בעזרת fetch&add, או להכריז על זה כקטע קריטי (ולעשות מניעה הדדית). אופציה נוספת היא לוותר על מקום אחד בבאפר- בבאפר מלא, לא נכתוב לתוך מקום שממנו נקרא. כך נוכל להבדיל בין הסיטואציות- קונפיגורציות.

פתרון אחר, בו נשתמש ב3 סמפורים, כאשר יש לנו יותר מקורא/כותב אחד (הפתרון הקודם לא עובד כאן).

מבנה נתונים נוסף- **Monitor**. הוא קצת יותר מסובך, דומה יותר לאובייקט- עם דאטה ופרוצדורות עליו. הוא דואג לזה שדברים לא יוכלו לקרות במקביל. נותן עוד דרך יותר קלה ויותר יעילה לכתוב ולהביע את אותם דברים. מאחורי הקלעים, הרבה פעמים זה ממומש ע"י סמפורים של מערכת ההפעלה. המניעה ההדדית מתקבלת בחינם. מפסידים קצת בביצועים ובמקבול.

את הבעיות נראה בהמשך.

י"א אייר ה'תש"פ, תרגול 6, עידן

4 תנאים שצריך שיתקיימו כדי שיקרה **דד-לוק**.

איך מערכת ההפעלה יכולה להתמודד עם דד-לוק? (זה אומר שמצבנו גרוע, כי אין לנו פתרון יחיד טוב לבעיה)

- להתעלם- גישת היען (אם תהליכים הכניסו את עצמם לבעיה, זאת בעיה שלהם. מקסימום יאתחלו את המחשב)
  - בזה מערכות ההפעלה משתמשות, הגיוני כי:
    - המחיר של האחרות יקר
    - זה יחסית נדיר ולא קורה
- detection and recovery – מאפשרת להכנס, אבל אחרי שמזהה מנסה לשבור אותו איכשהו
  - detection:
    - בניית גרף – הקודקוד הם הפרוססים, וקשת מפרוסס אחד לאחר אם אחר מחכה שהשני ישחרר איזשהו משאב. מתחילים מבניית גרף הקצאת המשאבים. מורידים את הקודקודים של המשאבים והקשתות ומחברים קשתות מתהליכים שהצביעו על משאב לזה שהמשאב מוקצה לו.
    - חיפוש מעגלים בגרף (אם יש מעגלים, יש דדלוק) – אלגוריתם מבוסס DFS. זמן ריצה במקרה הגרוע הוא n^2 כשn הוא מספר התהליכים. להריץ את זה פעם בכמה זמן זה overhead לא זניח.
  - recovery, 2 גישות:
    - להפיל את כל מי שגרם לדדלוק. יש לזה מחיר מאוד גדול.
    - ננסה להתחיל מלבטל משאב אחד. גם צריך לבחור את מי להרוג, בצורה רנדומלית או לפי עדיפות כלשהי? וצריך לבדוק אם הבעיה נפתרה.
- avoidance– אף פעם לא תתן למצב שבו יש דד-לוק לקרות, תמיד עם יד על הדופק
  - מימוש ע"י אלגוריתם הבנקאי ####
- prevention – תבטל את אחד מארבעת התנאים שראינו קודם, ולכן לא יוכל לקרות אף פעם. ננסה למצוא את מה מהתנאים נוכל לבטל
  - ביטול mutual exclusion: לאפשר לכמה תהליכים להשתמש באותו משאב. לא תמיד ישים.
  - ביטול no preemption: אפשר לקחת מתהליכים הקצאה של משאב בכח לפני שסיים להשתמש בו. יש מצבים שבהם זה אפשרי, ויש שלא. אבל לדוגמא מדפסת- לא הגיוני לקחת את המשאב לתהליך שמדפיס משהו. אפשר לעשות שאם תהליך מחכה למשאב מסוים, הוא משחרר את המשאבים שיש לו ומבקש אותם מחדש.
  - ביטול hold and wait: נדאג שתהליכים לא מחזיקים משאבים וגם מחכים למשאבים מתהליכים אחרים. הוא יוכל להתחיל לרוץ רק אם מלכתחילה יש לו את כל המשאבים לצורך הריצה (אם לא הצליח, משחרר אותם ומנסה להשיג הכל מהתחלה). אם הצליח, הוא ירוץ וישחרר הכל בסוף. בעיות:
    - ניצולת גרועה. יכול לקחת המון זמן
    - יכול לגרום להרעבה, והוא לא ירוץ אף פעם

- בעיה טכנית- לא יודע מראש מה יצטרך להמשך

מוסיפים לתהליך חותמת זמן של מתי הוא נוצר, ולכל משאב יש חותמת זמן של התהליך שבו הוא נמצא. ואז לפי החותמות זמן נוכל לבדוק אם ביקש אותו תהליך מוקדם יותר. לקרוא על זה עוד מהמצגת

- circular wait: למספר את כל המשאבים המשותפים (אינדקסים בסדר עולה), ובשתהליכים מבקשים משאבים מסודרים הם יבקשו אותם לפי סדר האינדקסים שלהם. לדוגמא פותר את בבעיית הפילוסופים, עם מספור הצ'ופסטיקים.

משתנים פרימיטיביים אטומיים std::atomic שיש לו 3 פעולות: load(),store() ואופרטור=.
מתנהג דומה למשתנה רגיל. זה לא יעבוד על מחלקות, ולא תמיד עדיף.
Multiple Fields לפעמים נרצה לשמור באותו משתנה כמה ערכים. לדוגמא נשתמש בint16_t ונדחוף פנימה מספרים עם דחיפה בכמות המקומות שנרצה a+=1<<8. אם נרצה לגשת רק ל8 הביטים הימניים לדוגמא, נעשה & עם מספר שמורכב מ8 אפסים ואז 8 אחדים. אם נרצה לגשת ל8 הביטים השמאליים, נעשה שיפט ימינה 8 ביטים a>>8.
נרצה להשתמש בזה במקום לדוגמא סטרקט שנרצה שיהיה אטומי.

למבחן צריך ממש לזכור את האלגוריתמים שלמדנו לפי שמות.

ט"ז אייר ה'תש"פ, הרצאה 7, דוד

מוניטור מוגדר בשפת התכנות, בניגוד לסמפור שהוא מבנה נתונים שמוגדר במערכת ההפעלה. יש לנו ממש קלאס שיש לו דאטה משותף ופונקציות שמאפשרות גישה אליו. המוניטור מבטיח שבכל רגע נתון רק טרד אחד יכול לגשת אל הדאטה המשותף. הוא הרבה יותר מגביל מאשר קטע קוד קטן- כי נועלים את כל האובייקט.
בג'אבה משתמשים במילה השמורה synchronized.
יש condition variable, כדי שכשטרד מחכה למשהו- הוא יעשו את זה מחוץ למוניטור. הפעולות עליהם:

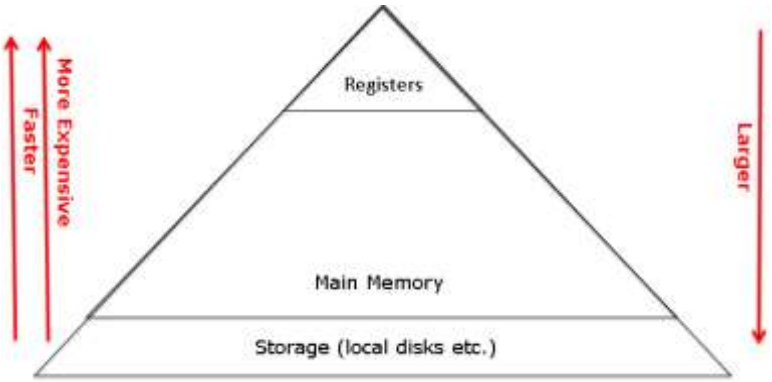
- wait(c)/c.wait() – משחרר את הנעילה ומחכה למישהו אחר שיעיר אותו. יש תור שבו הם מחכים
- signal(c)/c.notify – מעיר לכל היותר טרד אחד שמחכה. אם אין טרד שמחכה, הסיגנל נמחק
- broadcast(c)/c.NotifyAll() – מעיר את כל מי שמחכים

אפשר להוסיף למוניטור יותר מcondition variable אחד. אבל לפעמים אפשר לייעל ולהשתמש רק באחד, שמובנה במוניטור עצמו. לו נקרא בלי להגיד מה השם שלו.
הרבה פעמים נחכה אחרי בדיקה של תנאי כלשהו. נעשה אותו בwhile ולא בif, כדי לבדוק שוב את התנאי כשמשחררים. יש מצב ש"נפספס" סיגנלים ולכן נרצה לעשות notifyAll. לכן נצטרך לבדוק שוב את התנאי כי כמה יתעוררו יחד.

לפעמים המשאב המשותף נועד להחליף מידע בין 2 הטרדים/הפרוססים. זה מודל אחד של החלפת מידע ביניהם- זיכרון משותף. המודל השני, והמקובל יותר, הוא העברת הודעות ביניהם. עושים את זה דרך סוקט. רוב הבעיות נשארות זהות. הבעיות האלו נקראות אלגוריתמים מבוזרים.

היררכית הזיכרון

יש שלושה סוגים של זיכרון, מסודרים לפי הגודל שלהם:



הטכנולוגיות שבהן אפשר להשתמש:

- static RAM (SRAM)– מהיר, יקר (משתמשים לרגיסטרים)
- dynamic RAM (DRAM)– זול ואיטי יותר, אבל רגיש לרעשים. כל כמה מילי שניות (10-100) צריך לרפרש אותו, מה שמאט אותו. (משתמים לזיכרון המרכזי)
- יש טכנולוגיה שניה, שהיא בצורת דיסק- מעגלי (משתמשים להרד דיסק). הוא כל הזמן מסתובב. קריאת מידע לוקחת בין 5-9 מילי שניות.

ההבדלים בגישות הן בסדרי גודל בין אחד לשני.

יש זכרונות שהם נדיפים volatile – ברגע שהמחשב נכבה, הם נעלמים. רק הstorage הוא לא נדיף (הדיסק).

יש עוד סוג של זיכרון- ROM. לא חלק ממערכת הזיכרון. הכתיבה אליו מאוד איטית ומאוד קשה, ולכן הוא משמש בדר"כ לשמור בו קושחה, שלא מעדכנים באופן תדיר אלא פעם ב. עליו שמים את הBIOS, ואת המידע מה לעשות כשהמחשב נדלק.

מה cache מנצל? עקרונות הלוקאליות:

- לוקאליות בזמן- אם ניגשתי למקום מסוים בזיכרון, יש סיכוי לא רע שארצה לגשת אליו שוב בקרוב (x++, לולאות).
- לוקאליות במרחב- אם ניגשתי למקום בזיכרון, יש סיכוי לא רע שאגש אחר כך למקום 'קרוב' בזיכרון- לפניו או אחריו (גישה למערכים).

הרעיון הוא להחזיק קרוב יותר (זיכרון יותר מהיר) זיכרון שהCPU יצטרך בקרוב. הCPU cache ממומשים גם עם SRAM. אבל יש כמה רמות של זה, והזיכרון שונה. לרוב יש 3 רמות, כשכל אחת עם זיכרון אחר של SRAM, עם אותה חלוקה כמו שהיה קודם. L1 וL2 הם פרטיים, L3 הוא משותף. באותה צורה, הmain memory משמש cache של הstorage. כלומר כל שכבה היא קאש לשכבה שמתחתיה.

<u>הגדרות:</u>

- cache hit – חיפשתי משהו בקאש והוא היה שם. זה יחס- מספר ההצלחות/מספר הניסיונות
- cache miss – חיפשתי משהו בקאש והוא לא היה שם. אז יורדים לחפש ברמה הבאה, עד שמוצאים. גם יחס- מספר הכשלונות/מספר הנסיונות
- miss penalty – כמה זמן לוקח עד שסיימתי את החיפוש ומצאתי

ההאטה של הקצב (מ100% ל90% בL1) היא בערך ב50%, בגלל ההבדל בסדרי הגודל. במציאות, בדר"כ האחוזים הם 95%-97%.

הדילמות בתכנון קאש:

- איך מממשים את הגישה המהירה?
  - הגודל של הכתובת הרגילה גדול (64 ביטים לרוב), אבל הקאש קטן. איך נמפה את הגודל של ה64 ביטים למיקום האמיתי בקאש? נשמור במיקום לפי הLSB, ונשמור איתנו גם את הMSB (כל שאר הביטים, נקרא להם tag) כדי לדעת שמדובר באמת בכתובת הנוכחית. את הtag נשמור לדוגמא בבלוקים של 8, והפעולות שם יהיו פעולות על בלוקים ולא על מילים. ואז מורידים את הoverhead. מחלקים את הבלוק. ממספרים את הבלוקים ולפי החלקים ניגשים לחלקים.
- מה שומרים בו? מה האלגוריתם שמחליט מה נשמור.
- הוא בסופו של דבר יתמלא- אז את מה מוציאים מהקאש?
- מה קורה אם כותבים לאיבר שנמצא בקאש? איך מפעפעים את הכתיבה עד לזיכרון המרכזי- שהזכרון יהיה עקבי.

י"ח אייר ה'תש"פ, תרגול 7, עידן

לעבור קצת על המבוא שפספסתי, ועל החלק שהוא כתב עם הזיכרון (עד 17 דק)

לרוב בזיכרון נשמור מילים ולא ביטים או בתים.

לזכור את החזקות של 2 בעל פה.

מה קורה מאחורי הקלעים בcache? הכתובות שהCPU משתמש בהן הן כתובות ב main memory ולא בcache. אבל כשאנחנו קוראים מידע, אנחנו רוצים לבדוק אם הוא כן בקש. וגם אחר כך- להחליט לאן לכתוב אותו.

אף פעם לא נעתיק מילה יחידה לקש, אלא בלוק- כי הגיוני שאם ניגשנו לתא אחד, נרצה לגשת גם למלא שלידו.

שיטת הdirect mapping: כל בלוק ממופה לכתובת יחידה בקש: הכתובת מודולו מספר הבלוקים בקש. הכתובת היא כתובת הבלוק (אם הבלוק הוא בגודל מילה אחת, הכתובת זו הכתובת שלה. אם בגודל שתי מילים, נתעלם מהביט האחרון של הכתובות כדי לקבל את כתובת הבלוק).

האופסט (מה שהתעלמנו) זה כדי לדעת לגשת לתא הספציפי בתוך הקאש. מה שירד במודולו, נתייחס אליו כתג ונשמור אותו, כדי לבדוק איזה חלק מהזיכרון המרכזי נמצא פה (להבדיל בין כמה חלקים בו שממופים לאותו כתובת).

יש בעיות בשיטה הזו- כי כל בלוק יכול להתמפות למקום אחד בלבד, אז אם יש 2 בלוקים בשימוש נפוץ שמתמפים למקום זהה, הם הולכים להתחרות על המיקום הזה.

שיטת fully associativity: מוותרים על זה שכל בלוק יתמפה למקום יחיד. החסרון, שצריך לעבור על כל הcache כדי למצוא את זה. זה נמצא בחומרה, וזו בעיה לחפש את זה שם בצורה מהירה.

אז נשנה, שכל בלוק יוכל להתמקם לקבוצה של מקומות בקש.

שיטת 2^x-way associativity: הx משתנה. כל סט מכיל 2 באיקס בלוקים, ושומר את כל המיקומים האפשריים שבלוק מסוים מהזיכרון יוכל להתמפות אליו. ואז יש כמה אפשרויות למיקום, צריך לציין איפה נמצאים הבלוקים האלו (XX110 או 110XX).

נוכל לדעת אם שני תאים צמודים נמצאים יחד בבלוק בcache לפי האינדקס. כמובן שנכנס לcache הערך ולא הכתובת בזיכרון.

<u>אלגוריתמי החלפה</u>

אנחנו רוצים להוריד את הקצב של הפספוסים של הקש. לרוב, הוא כבר מלא ונצטרך להחליף תאים. **את מי נפנה מהcache?**

- שיטת FIFO: קלה למימוש (בחומרה) ולהבנה
  - חסרון- אין התייחסות לאם ניגשים לבלוק הרבה פעמים
- FIFO עם שדרוג: לכל בלוק נכניס ערך R ונשנה אותו ל1 כשניגש אליו שוב. עובד כמו FIFO, רק כשניגשים לבלוק שרוצים לשנות- אם R הוא 1, נשנה אותו ל0 ונעביר לסוף התור.
  - לא כל כך יעיל, עם הרבה תזוזות של הראש.
- שיטת balady's min- להדיח את מי שנצטרך עוד הכי הרבה זמן. לא ניתן למימוש כי אי אפשר לדעת מראש את זה. מדברים על זה עדיין כי היא אופטימלית ואליה ניתן להשוות שיטות אחרות.
- LRU- הדחת הבלוק שהשתמשנו בו הכי פחות לאחרונה. היא ישימה, והטובה ביותר שאפשר להשיג, כי לרוב לתוכניות יש התנהגות דומה. משתמשים בקאונטרים או במחסנית.
  - דורש תמיכה מיוחדת בחומרה.
- פסאודו-LRU – משפחת אלגוריתמים שמדמות את הLRU.

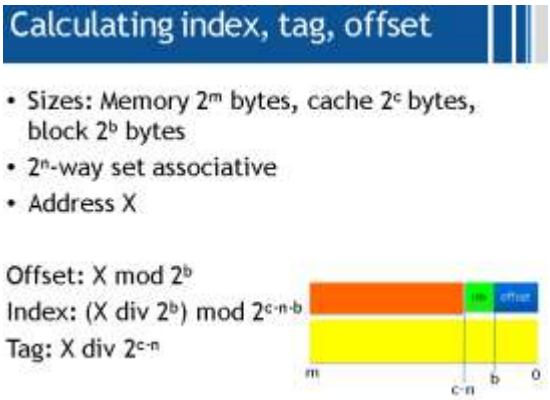
ב"ד אייר ה'תש"פ, הרצאה 8, נטע

נדבר על דילמות בcache.

<u>איך לממש גישה מהירה?</u>

חישוב מיקום בcache בשיטת 2^x-way:

- |offset|=cache block size : החזקה של גודל של בלוק בcache
- |index|=cache size/(cache block size+2^x) : החזקה של גודל הcache לחלק לגודל בלוק בcache ועוד גודל השיטה (יכול להיות גם 0)
- |tag|=|main memory address|-|index|+|x|-|offset| : מה שנשאר



בשיטת 2^x-way נוסיף מימין לאינדקס ביט (רק בבדיקה איפה נמצא) שמסמל לנו באיזו קבוצת בלוקים מדובר. כל הבלוקים של קבוצה ממוקמים באופן רציף בcache.

ככל שx גדל, גודל התג גדל. יש יותר תקורה בחיפוש הבלוק, אבל פחות בעיות שדברים לא נמצאים בו.

למה קורה cache misses?

- בהתחלה הcache ריק וכל דבר לא יהיה שם
  - נחשב זניח, כי מתמלא מאוד מהר
- גודל הקיבולת שלו
  - פתרון: להגדיל את הcache
- מיקומים רבים ממופים לאותו מקום
  - פתרון: הגדלת הcache
  - פתרון: שימוש באסוציאטיביות
- עקביות- תהליך אחר/פעולת IO עושה עדכון של התא בזיכרון
- שמירה בcache של מילים שלא הכי כדאי לשמור

<u>איך מחליטים את מי לשמור?</u>

הפספוסים יכולים לקרות במהלך כתיבה או קריאה. אם היה פספוס, לרוב הערך ייכתב לתוך הcache. ישמר לפי העקרונות של לוקאליות בזמן. בגלל העיקרון של לוקאליות במקום, ישמר כל הבלוק. נצטרך לפנות לו מקום.

בכתיבה, יש 2 אפשרויות:

- לכתוב את הערך למקום הנמוך ביותר בהיררכיה בו הוא נמצא. זה לא מעלה אותו לcache. לא צריך לקרוא את כל הבלוק אלא לעדכן את המילה.
- כמו בקריאה- נביא את הבלוק לcache ונכתוב את הערך לשם.

כתיבה היא כמובן מסובכת יותר, כי צריך לשמור על הערכים עקביים.

<u>את מי נעיף מהcache?</u>

בדר"כ הcache מלא במידע. לכן כשיש פספוס, נצטרך לפנות מישהו מהcache. הוא נקרא הקורבן. הוא צריך להיות בקבוצה שאליה מתמפה האיבר החדש.

איך בוחרים קורבן? אפשר רנדומלית, אבל עדיף לא להעיף בלוקים שמשתמשים בהם הרבה, כי יש סיכוי גדול שנרצה אותם בחזרה.

אלגוריתמים אפשריים:

- אופטימלי
  - להוציא את מי שיקח הכי הרבה זמן עד שנשתמש בו שוב
  - לא פרקטי למימוש, אבל נותן לנו חסם תחתון
- NRU (not recently used)
  - הערכת העתיד על בסיס העבר. לכל בלוק יש ביט, שנקבע ברגע שניגשים אליו. נאפס את הביטים בכל קריאת שעון (או אם ניגשנו לכל הבלוקים). כשצריך לבחור קורבן, בוחר רנדומלית מבין כל אלו שהביט שלהם הוא 0.
  - ניתן ליישם אותו, אבל הוא מאוד פשטני.
- FIFO
  - מיון הבלוקים לפי סדר כניסתם לcache (שמירה ברשימה מקושרת), הקורבן הוא הבלוק הראשון ברשימה.

- הבעיה היא שלא מתייחסים לתדירות השימוש, אלא לזמן כניסתם לcache. למרות שיכול להיות שהבלוק הותיק ביותר יהיה השימושי ביותר.

- second chance FIFO
  - נוסיף ביט שמאותחל ל1, והוא מאותחל ל1 גם כשקורית קריאה חוזרת לבלוק. זה פועל כמו FIFO, אבל עם הזדמנות שניה. בוחרים קורבן מראש התור- אם הביט שלו הוא 0, מפנים אותו. אם הוא 1- משנים אותו ל0, מעבירים לסוף התור ועוברים לבא בתור עם אותו הליך.
  - נחשב ללא יעיל כי מזיז בלוקים ברשימה המקושרת. כדי להתמודד עם זה, הוצע אלגוריתם second-chance clock שבו הרשימה המקושרת היא בצורת מעגל, ללא הזזת הבלוקים בcache- ולכן קל יותר למימוש.
- LRU (least recently used), pseudo-LRU – **הכי נפוץ וחשוב**
  - מבוסס על לוקליות בזמן- בלוקים שהשתמשנו בהם לאחרונה, הגיוני שנשתמש בהם שוב.
  - קשה ויקר למימוש. צריך לשמור את הזמן שבו ניגשנו אליו לאחרונה- אבל לא ברורה רמת הדיוק. צריכים למצוא את המינימלי בכל פעם- אז צריך מבנה נתונים יקר יותר.
    - אפשר למשש עם רשימה ממוינת, אבל אז צריך למיין בכל פעם. תקורת הזיכרון היא $\log_2(n)$ ביטים לכל בלוק.
    - מימוש אחר בחומרה הוא עם מטריצה בגודל nxn, כאשר n הוא מספר הבלוקים בcache. מאתחלים את המטריצה למטריצת הס. בגישה למיקום מסוים, הסמן את כל השורה שלו בו ואת העמודה בס. כשנרצה לפנות בלוק, נבחר את זה שבשורה שלו יש את הערך הבינארי הנמוך ביותר.
  - קשה לממש את זה, ולכן פותחו קירובים אליו. למשל שימוש בעץ עם תוספת של ביט של בקרה. הם מציינים איפה המקום היותר ותיק- 1 במקום הותיק יותר. זהו אלגוריתם זול יותר, כי שומר רק ביט אחד.
- LFU (least frequently used)
- Random
- ועוד

<u>ניהול זיכרון</u>

איך תוכנית רצה? היא תהיה חייבת לעלות לזיכרון.

חייבים לתרגם את הכתובות שתהליך מתייחס אליהן, לכתובות אמיתיות בזיכרון הפיזי, שבהן כתוב הדאטא שלו. כי הוא חושב שיש לו את כל הזיכרון, אבל זה לא נכון.

זה קורה גם בזמן הload לזיכרון. הרבה מהן נקראות relocated-כי משתנות בזמן ההעלאה לזיכרון. וחלק משתנות בזמן ההרצה.

כל תוכנית שמריצים חייבת קודם כל להיות בmain memory! לפחות שם, יכולה להיות גם בcache וכו', אבל ההוראות שלה חייבות להיות שם. לכן כל הכתובות שכתובות בקובץ exe שאותו רוצים להעלות, יצטרכו להיות כתובות שם.

יש חלק בCPU שנקרא MMU, שאחראי על תרגום של כתובות מנקודת מבט של התהליך- "כתובות לוגיות" לנקודת מבט אמיתית- "כתובות פיזיות".

בcache יש עוד כל מיני תרגומים של כתובות, אבל אנחנו מדברים על עוד לפני זה.
כבר דיברנו על תרגום בסיסי- base+x.
בעיות של הקצאה כזו: צריך להשאיר לו מקום לגדול כדי שהתהליך יוכל לעבוד כמו שצריך. יש הרבה "חורים" כאלו, שמאפשרים מקום לגדול, להגדלת הסטאק או ההיפ. אבל אי אפשר לשים שם דברים חדשים (כי זה חוטא למטרה). קוראים לזה Internal fragmentation. יכולים להיות גם חורים בתוך התהליכונים, או בצורת מימוש אחרת נעשה חור אחד לכולם.

- הקצאה רציפה לכל התהליכים (ברצף)
  - יכולה ליצור external fragmentation – רווחים בין תהליכים בגלל פינוי
  - הפתרון- להזיז ולסדר מחדש. לגרום לכל הזיכרון המוקצה להיות ביחד. הבעיות- יקר, ויוכל לתפוס הרבה מקום במעבר (להעתיק למקום אחר, לפנות ולהזיז).
- שימוש בדפים- לא נותנים לתהליך הקצאה רציפה, אלא מחלקים לדפים קטנים בגודל קבוע (לרוב 4KB).
  - צריך לעשות מיפוי בין דפים לאיפה הם נמצאים בזיכרון.

דף- החלק הלוגי שמחולק לקבוצות רציפות ממנו. מסגרת- חלוקה של המיקום בזיכרון הפיזי, בהם נשכן דפים. דפים ומסגרות יהיו באותו גודל- כדי שיכנס טוב.

- נצטרך שהMMU ידע את הטבלה- מיפוי בין דפים למסגרות. למעשה במקום הס של הטבלה יהיה המיקום של דף 0. כל טבלה כזו היא פר תהליך.

לתופעה כזו של שימוש בדפים קוראים pagigng. שומרת איפה החלקים הפנויים, וכשתהליך מתחיל ורוצה זיכרון, מנסים להקצות לו מקום מרשימת המקומות הפנויים, מפנים אותם מרשימת הפנויים וטוענים אליהם את המידע. כשהוא מסיים, נשחרר את המידע שלו כדי שאחרים יוכלו להשתמש.
תמיד מקצים הקצאות שלמות. הגדלה של גודל הדפים, מגדילה את הInternal fragmentation - כי אם מישהו ביקש 3.5 נקצה לו 4. מצד שני, הגדלה שלו מקטינה את גודל הpage table.
חישוב עם מספרים, ממחיש לנו שלא סתם נבחר 4KB להיות הגודל של דף.
בעזרת הטבלה ממירים את הכתובת- האופסט נשאר זהה, הMMU משנה את הביטים הראשונים- משל הדף לשל המסגרת (של הדף הוא האינדקס בטבלה לשל המסגרת).
הגודל של הpage table זה se/p- גודל של חצי דף. גם תופס מקום בזיכרון.
כל פעם שרוצים לגשת לזיכרון, צריך לעשות את ההמרה- כל גישה לזיכרון בעצם עולה פי 2.
הפתרון- cache מיוחד שמיועד רק לזה. TLB – translation lookaside buffer ברגע שאני יודעת מיפוי של מספר דף למספר מסגרת, אני שומרת אותו בcache (את שני פרטי המידע). ולכן כשאבוא לתרגם, אבוא קודם כל להסתכל בו (או במקביל). הוא cache מאוד מהיר- אם מצאתי את המיפוי שם ויתרתי על הזמן הזה. לרוב ממומש בfully associative, ואפשר לשמור בו רק 64 פרטי מידע.

כ"ו אייר ה'תש"פ, תרגול 8, איתן

גדלים: בשעושים חישובים עם גדלים שונים- להעביר לבתים ולא לשכוח להחזיר ליחידה שרוצים

- KB = 2^{10} bytes
- MB = 2^{20} bytes
- GB = 2^{30} bytes

עם x ספרות ניתן לייצג 2^x מספרים.

CPU יכול לבצע פעולות אריתמטיות בין רגיסטרים, ויכול להעתיק בין הרגיסטרים לזיכרון.
בנוסף, יש את השליטה על זרימת התוכנית.

כתובת פיזית בזיכרון (אמיתית/בינארית). האורך שלה הוא $\log_2$ של גודל הזיכרון.
אין לנו דרך לדעת מראש, איפה תוכנית תהיה בזיכרון הפיזי. בנוסף, אנחנו רוצים isolation- שתהליך אחד לא ייגש לזיכרון של תהליך אחר. הגודל הזה מוגבל. בגלל הisolation, כל תוכנית חושבת שהיא רצה לבד ושיש לה את כל הזיכרון.
כל הבעיות האלו נפתרות בעזרת מרחב הכתובות הוירטואלי. כשאנחנו מקמפלים תוכנית, כל הדברים שמקבלים כתובות וירטואליות בשלב הקומפילציה. המיקום בזיכרון הפיזי קורה בזמן ריצה.
איך זה מתבצע? מבוצע בחומרה ע"י MMU, שמתרגם מכתובת וירטואלית לפיזית.
<u>שיטה 1:</u> segmentation- מחלקים את התוכנית לכל מיני חלקים בגדלים שרירותיים. כל אחד מהם הוא מרחב כתובות וירטואלי קטן. אנחנו מסתכלים על מרחב הכתובות הוירטואלי, ושוברים אותו למספר סגמנט, ואופסט בתוך הכתובות. אנחנו מחזיקים טבלת סגמנטים, שבכל שורה בה יש את הכתובת ההתחלתית האמיתית שלו, והגודל שלו. יש לנו גם רגיסטרים, שמחזיקים איפה הטבלה נמצאת בזיכרון וגם גודלה. כשתוכנית מבקשת לגשת לכתובת מסוימת, היא תמיד מבקשת כתובת וירטואלית (כי זה מה שהיא מקבלת מהקומפיילר). בנוסף, יש ביט ולידציה, שבודק אם זה בכלל נמצא בזיכרון הפיזי. יכולים להיות ביטים נוספים- ששולטים על הרשאת קריאה/כתיבה/הרצה. כך נוכל להעביר מהRAM לדיסק.
לחזור לשקף 14
<u>שיטה 2:</u> נשתמש בpaging- חלוקה ל4 ביטים. לא יכולה להיות פרגמנטציה חיצונית- כל מסגרת תפוסה או פנויה. אבל יכולה להיות internal fragmentation, שזה פחות מטריד אותנו כי יבזבז לכל היותר חלק מדף אחד.
נצטרך מקום שישמור לנו את כל המסגרות הפנויות, כדי שמערכת ההפעלה תדע לאן למפות זיכרון חדש. גם כאן, האופסט נשאר זהה- רק מתרגמים את הכתובת של הדף לשל המסגרת, דרך הטבלה.
גם אותה- אנחנו צרכים לשמור איפה היא ומה הגודל שלה. גם כאן יש לנו ביטים נוספים- האם הדף ממופה למקום בזיכרון, האם עדכנו, האם השתמשנו (זה כמו cache, גם פה נצטרך לבחור את מי לזרוק וזה יעזור לבחור), הרשאות.
אם הcpu מנסה לגשת למקום בזיכרון שלא נמצאת שם פיזית כרגע, קורה אינטרפט מסוג page fault שגורם לתוכנית לרוץ, ומערכת ההפעלה תבחר לאן להכניס אותו. אחר כך, הוא יצא ממש של Blocked למצב של ready ויחזור על אותה השורה שקודם לא הצליח לעשות.
הבעיה היא שתהיה לנו טבלה מאוד גדולה, שברובה יהיה כתוב ששורות לא נמצאות בזיכרון הפיזי- מאוד לא יעיל. יש 2 גישות איך לפתור את זה:

- גישה הירכבית
- טבלאות הפוכות

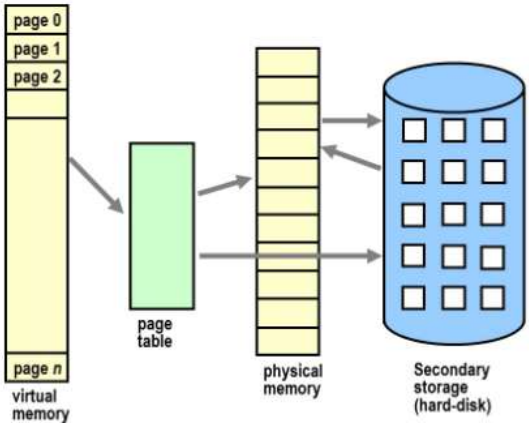
הגישה ההיררכית שוברת את מרחב הכתובות הוירטואליות לעוד טבלאות מיפוי, כלומר כמה שלבי מיפוי. בכל שלב מיפוי שבו יש לנו טבלה שכל הvalidation bit הם 0 (כלומר מכיל ערכי זבל), בכלל לא נשמור את הטבלה.

לדוגמא ב2 רמות, טבלה שניה עם ערכי זבל לא תישמר ולכן המקום היחיד ש"נבזבז" עליה יהיה שורה יחידה בטבלה ברמה הראשונה.
ככל שיש יותר רמות בטבלה, היא תופסת פחות מקום בזיכרון, אבל עולה יותר בגישות לזיכרון (יעילות זיכרון לעומת יעילות זמן ריצה).

א' סיון ה'תש"פ, הרצאה 9, דוד

זיכרון וירטואלי

הזיכרון הלוגי הוא הזיכרון שתהליך חושב שיש לו. אם הוא יודע מה גודל הזיכרון, זה יוצא גדול יותר מהזיכרון הפיזי. מצד שני, לפעמים מגבילים אותו לחלק מהזיכרון ואז לפעמים זה לא המצב.
מספר הפרוססים לא מוגבל, אולי רק במשהו שהוא כמעט אינסופי.
הזיכרון הלוגי של כל התהליכים יחד, גדול יותר מהזיכרון הפיזי.
כדי לעמוד בקונפליקט הזה, יש לנו זיכרון וירטואלי. זה אומר שרק חלק מהדברים של התהליך יהיו באמת בזיכרון הפיזי (השאר יהיה בדיסק, או בSSD), רק אלו שצריך יותר עכשיו. הזיכרון המרכזי הוא סוג של cache של הדיסק.



אם דף מסוים לא נמצא בטבלת הדפים, נצטרך להעתיק את הדף מהדיסק. זה לוקח המון זמן, וחוסם את התהליך הספציפי בינתיים.
אנחנו מוסיפים ביט נוסף לטבלת הדפים שאומר עם הדף הספציפי נמצא בזיכרון הפיזי או לא.
<u>הזיכרון הוא cache של הדיסק, אז צריך לענות על:</u>

- איך מדברים על גישה מהירה?
  - תרגום כתובות (טבלת כתובות וTLB)
- את מי צריך לשמור בcache?
  - רק כשמבקשים
  - (פחות שימושי) הos מנסה לנחש במי נשתמש. חוסך זמן אם הניחוש נכון (בי אפשר להעלות לאט, בלי לחץ ולתקוע דברים אחרים), אבל אם הוא טועה יהיו יותר פספוסי cache.
- את מי צריך להעיף מהcache?
  - משתמשים באחד האלגוריתמים שכבר ראינו. יש אלגוריתם אחד ספציפי (שנממש ב4): Working Set, WSClock שדומה לניסיון השני של FIFO.

בשדף הוא לא בזיכרון, קורה page fault. התהליך מוותר על הCPU בזמן שהדף נטען.

ההעתקה מהדיסק לזיכרון המרכזי קורית בעזרת הDMA ולכן לא צריך לחכות לו עם הCPU, אלא הוא יעדכן כשיסיים, ואז נעדכן את טבלת הדפים.

Effective access time = p(page fault time)+(1-p)(memory access time)
Slowdown = Effective access time / memory access time

Typical numbers:
page fault time: 25 ms
access time: 100 ns

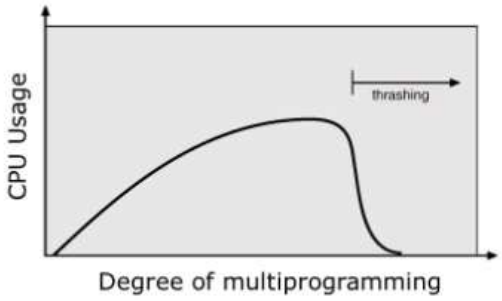
p=0.001 → slowdown = 250
p=0.0000004 → slowdown = 1.1

מה המחיר של פספוס דף?

בדר"כ נשאף להסתברויות נמוכות לקבל פספוס. נוכל להשיג את זה כי עובדים על דפים ולא על מילים בודדות, שהם מספיק גדולים. ברגע שאלגוריתם ההחלפה שלנו הוא טוב, זה יהיה טוב יותר.

CPU Usage: הזמן שבו הCPU עסוק בלעסוק הוראות. במולטיפרוגרמינג אנחנו רוצים להעלות את הcpu usage.

Thrashing אם יש לנו המון פרוססים, יהיה לנו יותר page faults והCPU כל הזמן עסוק בלהחליף דפים.



למה paging עובד? שימוש בלוקאליות במרחב (כמות הדפים שצריך להשתמש בהם ביחידת זמן).
למה thrashing קורה? גודל הלוקאליות (השילוב של כל אזורי העבודה של התהליכים השונים) > גודל הזיכרון הכולל.
הפתרון במקרה כזה, נוסיף מצב שנקרא suspended block, הגבלה של המולטיפרוגרמינג. לא ניתן ליותר מX תהליכים לרוץ, ונחליף ביניהם בקצב איטי יותר מאשר מחליפים בין תהליכים בנורמה.

כדי לפתור את הבעיה של טבלת דפים גדולה:

1. טבלה לפי היררכיות (שבירה לתתי טבלאות, וכך את מה שריק לחלוטין לא נשמור)

- מה שדיברנו בתרגול- מורכבת מטבלאות גדולות שטבלה גדולה מצביעה עליהן. מבנה של עץ. באבא כתוב באיזה מקום נמצאת הטבלה הבאה, בטבלה הבאה כתוב איפה נמצא הדף (בשיש 2 רמות).
  כל היררכיה מיוצגת ע"י לוג הגודל שלה ביטים, וכל השאר זה האופסט.
- במקום טבלה על מיליון כניסות (לזיכרון בגודל 2^{32}), עכשיו יש לנו 1001 טבלאות שכל אחת מהן עם אלף כניסות. היתרונות: לא צריך לשמור את כולן כל הזמן (וחלק לפעמים לא צריך לשמור בכלל), ולא צריך לשמור אותן באופן רציף.

לעבור על החישובים

- בהנחה שהטבלה הראשית בזיכרון, יכול להיות שיהיו 2 page faults. או שצריך להקצות את המקום הזה בזיכרון, עבור עמודים שריקים לגמרי.

2. טבלאות hash

- הרבה פחות פופלרית. צריך להחזיק פונקציית האש, ואז מחזיקים רשימה מקושרת בתא של מי ממופה לאן.

אם מגיעים לסוף השרשרת ולא מוצאים, יש page fault – אבל לפני זה אנחנו עוברים על רשימה מקושרת שיכולה להיות גם מאוד ארוכה. זה יכול לקחת הרבה מאוד זמן, הכל תלוי בגודל של טבלת ההאש ביחס לגודל הזיכרון הפיזי.
הוא יהיה צריך להיות כתוב באופן רציף בזיכרון.

3. Inverted – שמסתכלים על דברים הפוך

- בניגוד לשאר השיטות, פה תלויים בגודל הזיכרון הפיזי (כשהוא ממש קטן זה ישתלם). ואז הMMU שומר טבלה של מה נמצא בכל מקום בזיכרון הפיזי (לפי מספר תהליך ומספר העמוד בכתובות הלוגית).

אם הדפים נמצאים בTLB, נוכל לדלג על כל המסלול האיטי של הטבלאות ההיררכיות.
אחרי התרגום של הכתובת, נחפש את המסגרת קודם כל בcache (נתעלם מהחלוקה לכתובת דף ואופסט, ונחלק לתג, אינדקס ובייט), ורק אם יש cache miss ניגש לזיכרון המרכזי.

1. אם יש פספוס TLB, זמן הגישה יהיה ארוך בכל מקרה.
2. אם אין, כנראה שמצאנו את זה מאוד מהר ונרצה עדיין להמשיך מהר ושתהיה האצה.

האינדקס נמצא בתוך החלק של האופסט, ולכן נוכל לתרגם במקביל. ואז להשוות בין התגים שחילצנו מהTLB לבין החלק של המסגרת שקיבלנו.
אם באחד מהם יש cache miss נצטרך ללכת לזיכרון הראשי בכל מקרה.

Problem	Solution
Process view is different than actual view	Address Translation: base+bound, **page tables**
External fragmentation	Compaction, **paging**
Logical memory is larger than physical memory	**Virtual memory + swapping**

בעיות ופתרונן: (אקסטרנל פרגמנטשיין
זה פיצ'ר ולא באג, כי רוצים לתת לתהליכים מקום לגדול אליו)
כל חלק בא לפתור בעיה אחרת.

הmain memory הוא סוג של cache לדיסק. אז יש לנו את אותן הבעיות פה ועל זה דיברנו.
אלגוריתם החלפה working set – לוקאליות בזמן של הזיכרון, לכמה עמודים הוא התייחס בk הגישות האחרונות לזיכרון. צריך לעשות לk טיוב- שלא יהיה גדול מדי או קטן מדי. הרעיון הוא להבין על איזו קבוצה הוא עובד, ואותה לא לפנות. האלגוריתם שלנו יהיה לבחור עמוד שלא נמצא בworking set הנוכחי.
בשביל לממש, נצטרך לזכור מתי הייתה הפעם האחרונה בה ניגשנו לדף מסויים.

יותר קל להסתכל על הזמן- את מה עשיתי בk המילישניות האחרונות.

WSClock: משלב את הרעיונות של working set, ניסיון שני של FIFO וNRU כך:

- שומר את כל הדפים של התהליך שנמצאים בזיכרון הפיזי במעין מעגל, עם חץ שמצביע על איפה אני נמצא.
- יהיה משתנה של הזמן הוירטואלי (שבו ניגשתי לשעון)
- כל פעם שניגשים לעמוד מסוים, משנים את הרפרנס ביט שלו מ0 ל1. זה אומר שלאחרונה ניגשנו אליו.
- פעם בכמה זמן יש פסיקת שעון- בו עוברים על כל הדפים, ובאלו שהרפרנס ביט הוא 1 משנים את הזמן לזה של השעון, ומחזירים את R להיות 0. לזה שיש להם R=0 לא עושים כלום.
- ברגע שיש page fault:
  - מסתכלים על הדף שעליו מצביע החץ:
    - אם r=1 זה אומר שהשתמשתי בו לאחרונה, ולכן לא נפנה אותו ונקדם את החץ ב1
    - אם R=0 וגם הזמן הוירטואלי הנוכחי פחות הזמן האחרון בו השתמשנו בו קטן מK, נקדם את החץ
    - אם הדף מלוכלך, תקדם את החץ (אם מלוכלך אבל זקן יותר מK, נגיד לDMA להעתיק אותו לדיסק במקביל)
    - אחרת, תפנה את הדף
    - אם לא מצאתי אף אחד, ננסה לפנות את הדף הכי ישן
- <u>דף מלוכלך</u> זה דף שמאז שרשמתי אותו בפעם האחרונה לדיסק, עשיתי שינויים. כלומר ברגע שארצה לפנות אותו, קודם אצטרך לכתוב את התוכן שלו לדיסק.
  - לכן תמיד נעדיף לפנות דפים נקיים.

פינוי דפים לוקאליים- כל תהליך מפנה מעצמו. פינוי דפים גלובליים- אפשר לפנות גם מתהליכים אחרים. תלוי במערכת ההפעלה.

ג' סיון ה'תש"פ, תרגול 9, איתן

מערכת הקבצים:

מה זה קובץ? רצף של 0 ו1. מבנה אבטרקטי- יש לו שם, תוכן, metadata (מתי נוצר, מה הגודל שלו), ניתן לבצע עליו פעולות (לכתוב, לשנות שם).

אם מכבים את המחשב, הקובץ עדיין קיים. תהליך יכול להשתמש בקובץ, למות, ואז תהליך אחר ישתמש בו.

file metadata: השם הוא לא חלק, קובץ לא שומר את השם שלו אלא רק התיקיה שומרת את השמות של הקבצים שבתוכה.

- גודל
- owner
- הרשאות- קריאה/כתיבה/הרצה
- חותמת זמן- יצירה/ הזמן האחרון שעודכן/ניגשנו
- מיקום- בדיסק
- סוג- בינארי או טקסט (שתכלס גם הוא בינארי), האם הוא רגיל או תיקיה.

- אינדקסים היררכיים לבלוקים בדיסק בהם הקובץ שמור (בערך 12 ישירים, 1 indirect, 1 double indirect, 1 triple indirect).

בUNIX זה בנוי על סטרקט שנקרא inode שבו הולך להיות שמור כל המטאדאטה.

תיקיה (נשמרת כמו קובץ) זה סוג של קובץ פשוט, שבו כתובים שמות של קבצים ותיקיות שנמצאים בתוך התיקיה, ומה המספר Inode שלהם. חייבים להתחיל מתיקיית השורש, בלינוקס היא נקראת /. ניתן לראות את המספרים שלהם בעזרת ls –i.

איך מנהלים את הזיכרון? superblock מנהל את זה, מכיל:

- גדול מערכת הקבצים (FS)
- רשימה של הבלוקים הפנויים בFS
  - מוקצה האחד הפנוי הבא
- רשימה של inodes לא בשימוש (מספר הinode שיש הוא קבוע מראש, אפשר לשנות רק בשמפרמטים את המחשב)
  - בשinode משוחרר, המיקום שלו נכתב לכאן – רק אם יש פה מקום
  - אם הרשימה ריקה, הקרנל עובר על הדיסק ומכניס inode פנויים אחרים לתוכה
- ועוד

בעזרת המידע הזה אפשר להקצות בלוקים בדיסק בשביל לשמור file data וfile metadata.

שמירת מידע בקובץ דורשת:

- הקצאה של בלוקים בדיסק לקובץ
- אפשרות קריאה וכתיבה
- אופטימיזציה- להמנע מגישה לדיסק באמצעות cache למידע בזיכרון.

(FD) file descriptor זה מה שממש את הגישה האבסטרקטית לקובץ ולמשאבי IO אחרים. תהליכונים משתפים את טבלת הFD (והאופסט של הקבצים). ברגע שיש FD, הוא תמיד מצביע לאותו קובץ. בUNIX מסתכלים על הכל כקובץ. גם לסוקטים, לשליחת מידע באינטרנט. פתיחה מוצלחת של קובץ מחזירה FD (מספר אי שלילי, לכל תהליך יש טבלה שנקראת file descriptor table שמכילה אינדקס לטבלה הזו).

להשלים את מה שקרה בין 27 ל33

י' סיון ה'תש"פ, הרצאה 10, דוד

מערכת הקבצים

אפליקציות יכולות לשמור דברים בזיכרון המרכזי, אבל לפעמים זה לא טוב

- כי הגודל של הזיכרון המרכזי מוגבל.
- זיכרון נדיף- ימחק כשנצא מהאפליקציה (כשתהליך נגמר או כי ניתקנו מהחשמל)

לכן אנחנו צריכים מערכת

- שתוכל לשמור הרבה מידע
- המידע יוכל לשרוד את הפרוסס שמשתמש בו (גם את התהליך נגמר, בשביל תהליך אחר או המחשב נכבה)
- שהרבה תהליכים יוכלו לגשת למידע

  זו גם דרך לשתף מידע בין תהליכים

חלק מבעיות סינכרון בין תהליכים נפתרת ע"י מערכת האיחסון- כתיבה לתוכה כדי שהרבה יוכלו לקרוא משם.

צריך שזה יהיה גדול ולכן מדברים על הרד-דיסק או SSD. אנחנו משתמשים בדיסקים כדי לשמור מידע, ומארגנים אותו באבסטרקציה שנקראת קבצים, שנשארת גם אחרי שהמחשב נכבה ונמחק רק כשאמרנו לו במפורש. חלק חשוב מזה הוא ניהול הרשאות.

מערת הקבצים זה איך מערכת ההפעלה מנהלת את הקבצים:

- אינטרפייס של מערכת הקבצים
  - API, sys calls – כתיבה, קריאה, פתיחה. מיקום הקובץ- ארגון שלו בתוך ספריות. איך לשתף קובץ על תהליך אחר, ולהגן עליו מפני תהליכים אחרים.
- מימוש של מערכת הקבצים
  - איך שומרת? איך מנהלת? איך גורמת לשיתוף פעול בין תהליכים שונים? איך מאפטמת את הגישה לקובץ? דרך לאחזר קובץ.

<u>קובץ</u> היא יחידה לוגית של מידע שנשמרת על מערכת איחסון, שיכולה להכיל כל סוג של מידע- תוכנית (בינארי או קוד בשפה עילית) או כל דבר אחר.

OS נותנת ליצור, לכתוב, לקרוא, להעביר את הראש הקורא למקום אחר בקובץ, לחתוך את הקובץ, למחוק. כל זה קורה דרך system calls לOS (שתפעל בקרנל מוד), כדי שלא ניגש ישירות לדיסק ונעשה שטויות.

<u>מה קורה מנקודת המבט של המשתמש?</u>

- שמות לקבצים. שם ניתן בזמן יצירה של הקובץ. בדר"כ זו גם הדרך שלנו לתת גישה לקובץ לתהליכים אחרים.
  - עד 255 תוים
  - ניתן להשתמש בספרות ובתווים מיוחדים (תלוי מערכת הפעלה)
  - יש מערכות שבהן זה משנה אם זו אות גדולה או לא (ביוניקס משנה, בדוס ובוינדוס נחשב אותו דבר).

  זה קורה במקום שונה למקום שבו נשמר המידע של הקובץ.
- יש לקובץ כל מיני נתונים עליו. מה נשמר מאוד משתנה בין מערכות הפעלה. דוגמא לנתונים: שם, מזהה ייחודי, סוג (בינארי, אסקי, ...), מיקום תחילת הקובץ בדיסק, הרשאות, גודל, מתי נוצר, מי ניגש אליו לאחרונה.
- סיומת הקובץ שמצביעה על סוג – חובה רק בוינדוס ובדוס (אפשר לכתוב את זה גם ביוניקס, אבל זה לא מצביע על טייפ מבחינתה).
- ניהול הקבצים וארגונם- בתוך ספריות, מה שנותן:
  - יעילות מציאת קבצים
  - שמות נוחים למשתמש
  - ניהול בקבוצות לוגיות
- directories- בדר"כ בנוי בצורת עץ (המטרה היא להבין איפה נמצא הקובץ, כדי לקבל מצביע למידע). כל שורה בספריה יכולה להיות ספריה או מידע. כדי לשנות שם של קובץ לא צריך להגיע למידע שלו בדיסק, אלא רק למטאדאטא שלו- איפה שנמצא השם והמבנה של הספריה.

השם של הקובץ זה המסלול האבסולוטי מהשורש עד אליו, לפעמים נתייחס למיקום רלטיבי (כי המערכת יודעת איפה אנחנו עכשיו, ואז ניתן לעשות את זה עם גישה של \. או \..).

- לא תמיד זה באמת עץ, לפעמים יכול להיות שנצביע משני מקומות שונים לאותו מידע בזיכרון (לאותו תוכן של קובץ). לזה קוראים hard link. ניתן לעשות את זה עם ln בשל.
- symbolic link (or soft link) זה הצבעה כפולה לנתיב של קובץ או תיקייה (כלומר ניתן להצביע לדיירקטורי). ואז צריך לוודא שאין מעגלים בגרף. ניתן לעשות את זה עם ln –s. (ואם מישהו מוחק את הקובץ אין לנו יותר גישה אליו)

בעיה בגישה הקשה- אם מוחקים את המידע עצמו. אז נשאר מצביע באוויר (כי לא שמרנו לאן הוא מצביע). זה קורה. פתרון זה לשמור את מספר המצביעים- ואז אם מישהו מוחק, רק המצביע נמחק ולא הקובץ, אלא אם הוא המצביע האחרון.

<u>פקודת Mount</u>- איחוד של מערכת קבצים שלמה לתוך מערכת קבצים שלנו, כדי שנוכל להשתמש בה. מאוד שימושי (הכנסת USB, מכניס למיקום בתוך הזיכרון שלנו). ניתן לעשות את זה למשהו לוקאלי (דיסק און קי), או למשהו שנמצא במקום מרוחק.

- צריך לזכור שזה עדיין לא נמצא באמת בתוך מערכת הקבצים שלנו, ולוקח יותר זמן גישה.

<u>הגנה של קובץ</u>- לרוב ניתן למי שיצר את הקובץ להחליט מה אפשר לעשות, ומי יכול לעשות מה. לפעמים נותנים אותם ספציפית לכל יוזר בנפרד, או על קבוצות של משתמשים.

- יוניקס מסתכל על זה כשלושה תווים- וממיר למספר בינארי, עם התייחסות להרשאות ל3 קבוצות- אני, קבוצה, כל העולם. הפקודה היא chmod 716 לדוגמא (7 לי, 6 לקבוצה, 1 לעולם).
  - היתרון- קצר. החסרון- לא נותן הרבה כושר הבעה- יש 3 סוגי הרשאות ו3 סוגי קבוצות.
- וינדוס לכל קובץ כותבים מי יכול לכתוב מה.
  - היתרון- יש הרבה כושר הבעה. החסרון- קשה לעקוב ולוודא שנתנו הרשאות נכונות.

<u>קריאת מידע</u>

- מההתחלה לסוף. כל פעם שנרצה לקרוא עוד חלק, נטען אותו. לפעמים נצטרך לקפוץ אחורה. לדוגמא סרט.
- random access- המידע מפוזר (אין משמעות לסדר), ולכן נוכל לגשת בצורה כזו. לדוגמא מערכת מידע.

אנחנו צריכים לחשוב על:

- מה הDAST הכי טוב לשמור את המידע של מיקום הקובץ?
- איך לשמור את המידע בדיסק?
- צריך לאפשר גישה לפי הסדר ותמיכה בגישה רנדומית. (אחד עולה על השני)

בעיות: שברור (fragmentation), כמו בזיכרון. קובץ שנמחק משאיר חור שלא מספיק לקובץ הבא.

- פתרון- compaction- יקר ולא יעיל.

- פתרון- paging. שוברים את המידע לבלוקים, במקום לשמור בצורה רציפה.
  - היוזר לא מודע לזה, זה נראה לו כמשהו רציף
  - API- רצף של ביטים
  - בתוך הos- אוסף של בלוקים

<u>מנקודת המבט של הOS:</u> אוסף של בלוקים. ה**בלוקים** הם היחידות הלוגיות ש נשמרים על גבי הדיסק בתוך **סקטורים**. בדר"כ 4KB. כדי לקבל רק בתים מסוימים, קוראים בלוק שלם וממנו בתים מסוימים.

איך הם באמת מסודרים? צריך לנהל את איפה יש מקום ריק. הדרך לעשות את זה תלויה במספר גורמים: לתמוך בגישה לסוגי קבצים שונים, באיזו מדיה הם נמצאים, איפה המטא-דאטה נשמר ואיך (יחד או בנפרד מהקבצים), ודברים שקשורים למערכת ההפעלה.

ההאדר-דיסק (הוא עדיין השמירה הנפוצה יותר) בנוי מפלטות שכל הזמן מסתובבות. כשרוצים לגשת למקום מסוים, צריך להעביר את הזרוע המגנטית למסילה הנכונה ואז בסיבוב שיגיע למקום הנכון (rotational time). להזזה מטראק לטראק קוראים seek time שנמדד במילי שניות.

הסיבוב של הדיסק נמדד בסיבובים לדקה RPM. ההעתקה של המידע קורית מעל הבס לזיכרון המרכזי ולוקח עוד 8-12 מילישניות.

היינו רוצים שדברים יהיו אחד אחרי השני, כדי שנוכל לקרוא מהר ולא לחכות הרבה זמן. או לפחות שיהיה באותו טראק או טראקים קרובים. אבל לא נשמור אותם רציף כי זה יגרום לבעיה של פרגמנטיישן. מה נעשה:

- בלוקים ישמרו לנו בלינקד ליסט, שבסוף כל בלוק יהיה המיקום של הבלוק הבא. הבעיה:
  - המידע שאפשר לשמור הוא לא חזקה של 2 (כי גם הפוינטר לוקח מקום) ודברים יכולים להתבלגן.
  - טוב לקריאה לפי הסדר, אבל random access איטי ממש- נצטרך לעבור הכל בדרך.
- שמירת טבלה חיצונית FAT ששומרת איפה כל קובץ נמצא, וכל אחד כותב איפה נמצא הבלוק הבא עד ל-1 שמצביע שזה סוף הקובץ. גודל הטבלה הוא מספר הסקטורים (הבלוקים הפיזיים).
  - יתרון- הטבלה שמורה בזיכרון המרכזי ולא צריך כל פעם לקרוא בלוק מהדיסק. זה חיסכון ב3-4 סדרי גודל. הניהול לא בדיסק.
  - יתרון- random access הרבה יותר מהיר
  - חסרון- יכול להיות מאוד גדול. אפילו אם שומרים מעט זיכרון.
  - בזה משתמשים בוינדוס.
- פתרון ביניים- טבלה שלא כולה נמצאת בזיכרון המרכזי. indexed allocation- כל קובץ מקבל אינדקס התחלתי, שאומר מה המיקומים של הבלוקים של הקובץ (בגודל מוגדר).
  - בUNIX שמרו באינדקס הזה כבר את כל המידע של הקובץ (בלי השם). זה מוסיף גישה נוספת לזיכרון בגישות הרגילות (עם החלוקה הקודמת, כשמה שרחוק יותר ודורש יותר יהיה מה שנשתמש בו פחות).

י' סיון ה'תש"פ, תרגול 10, עידן

תקשורת

מילת המפתח בנושא של תקשורת זה **פרוטוקול**. זו מוסכמה בין שני צדדים שמדברים אחד עם השני, שצריכים לציית לחוקים כדי שהתקשורת תתקיים. במחשבים צריך שהכללים יהיו מוגדרים היטב. יש כמה שכבות של פרוטוקולים, שכל שכבה אחראית על סוגיה אחרת.

דוגמא- מייל:

- תמיד יכול להשתבש משהו בדרך (וככל שההודעה גדולה יותר, יש יותר סיכוי שזה יקרה).
- במקום להגביל את אורך ההודעה, נפצל אותה לחלקים קטנים- packets, שנצטרך להוסיף גם מידע לגבי סדר הפקטות (overhead קטן). זו שכבה של מידע שנוספת למייל, שצריך להוסיף בשביל תיאום. אבל יכול להיות שבדרך פקטה אחת תעלם, או שהן יגיעו בסדר הפוך, או שאחת תשוכפל בדרך.
- בדר"כ ההודעה לא תעבור ישירות כי אין לנו חיבור ישיר למי שנרצה להעביר אליו, אלא תעבור בתחנות בדרך. ונצטרך לכל פקטה לצרף את הכתובת של הנמען.
- יכולת לתקן שגיאות. נוסיף עוד שכבה קטנה של ECC, שאיתו נוכל לשחזר עד רמה מסויימת את הפקטה המקורית במידה והיא השתבשה.

טבלה של השכבות שיש באינטרנט במצגת, שלכל אחת יש פרוטוקולים משלה.

Internet protocol suite (TCP/IP) –
The protocol stack that is used by the Internet

11

Layer name	Description (Layer's goal)	Protocols
Application	process-to-process communications	HTTP/S, SSH, FTP, DNS
Transport	End-to-end communication services for applications	TCP, UDP
Network / Internet	Transport datagrams (packets) from the originating host across network boundaries, if necessary, to the destination host specified by a network address	IP
Link / Physical	Communications protocols that only operate on the link that a host is physically connected to.	802.11 WiFi, Ethernet

<u>שכבת הtransport-</u> אחראית על השידור ולפתור בעיות של אמינות. יש 2 משפחות של פרוטוקולים.

פורט- כותב בכתובת היעד את הIP של המחשב שאליו אנחנו מעבירים, אבל איך נדע לאיזה תהליך אנחנו מעבירים את המידע? כל תהליך שרוצה קשר עם העולם החיצוני יוסיף לעצמו מספר פורט, ונשלח אליו יחד עם מספר פורט.

UDP- דואג לטיפול בבעיות בצורה מינימלית. לא מוסיף מספר פקטה, ככה שהסדר של הפקטות יכול להשתמש. הוא לא מוודא עם הצד השני שהוא קיבל את המידע כמו שצריך.

TCP- מוודא שהצד השני קיבל כמו שצריך. הצד המקבל שומר אצלנו את הדברים בבאפרים. מטפל בבעיות של איבוד פקטות, סדר שלהן, כפילויות ועיכובים.

Property	UDP	TCP
Reliable	no	yes
Connection type	Connectionless	Connection oriented
Flow control	No	Yes
Latency	Low	High
Applications	VOIP, Most games	HTTP, HTTPs, FTP, SMTP, Telnet, SSH

Sockets

אובייקט שמנהל את העברת המידע בצורה של client/server.

התהליכים יוצרים את זה ומשתמשים בזה ישירות. מערכת ההפעלה היא זו שמנהלת את זה מאחורי הקלעים. צריך להיות בשני הצדדים, ומערכות ההפעלה בשני הצדדים דואגות לזה כדי שתהיה העברה של מידע.

יש שני סוגים שמתאימים לשני סוגי העברת המידע.

הסרבר יוצר סוקט שמחכה לקליינט שמגיע. הקליינט גם צריך סוקט משלו. צריך לציין את הIP והפורט של התהליך והמחשב שאליו הוא מתחבר. הם מגדירים מה סוג התקשורת ביניהם. הסרבר יוצר סוקט חדש עבור כל קליינט שמגיע, מאותו סוג תקשורת.

סרבר יכול להתחבר לכמה קליינטים.

stream- לא קשור דווקא לרשתות, אלא להעברת מידע בין תהליכים. קשור להעברת ביטים בין תהליכים, ממקור חיצוני (אפילו דרך המקלדת מהמשתמש).

מימוש בפועל

sockets' address יש סטרקט שממומש בשפה ומיועד לזה.

יש לנו 2 דרכים לשמור מספר שמורכב מכמה בתים בזיכרון- מהLSB לMSB או להפך. אין העדפה לאף אחת מהן. השמירה במחשב היא מLSB לMSB, אבל התקשורת היא הפוכה- ולכן נצטרך לדעת להמיר בין 2 האופציות לכל סוג מספר שנרצה לקרוא.

הFS מתייחס לסוקט כקובץ רגיל, ויש לה גם FD שמתקבל כשיוצרים אותו.

DNS פרוטוקול שדואג לתרגם מכתובות הנוחות לנו לכתובות IP.

sockets programming – נמצא במצגת.

טיפול בהרבה קליינטים- פונקציית select.

ט"ז סיון ה'תש"פ, הרצאה 11, דוד

יש משחק כפול של קאשינג בין הmempry main לדיסק- נשמור בדיסק דברים מהזיכרון הוירטואלי שאין לנו מקום אליהם בmemory main, ונשמור בזיכרון המרכזי את מה שקראנו לאחרונה מהדיסק (שזה קאש לדיסק, במקום מוגדר בו).

מערכות קבצים ניתן לחלק ל2 סוגיות:

- אינטרפייס- מה הOS חושפת לאפליקציה
- מימוש- בתוך מערכת ההפעלה, ואיך היא מנהלת את הכתיבה על הדיסק עצמו

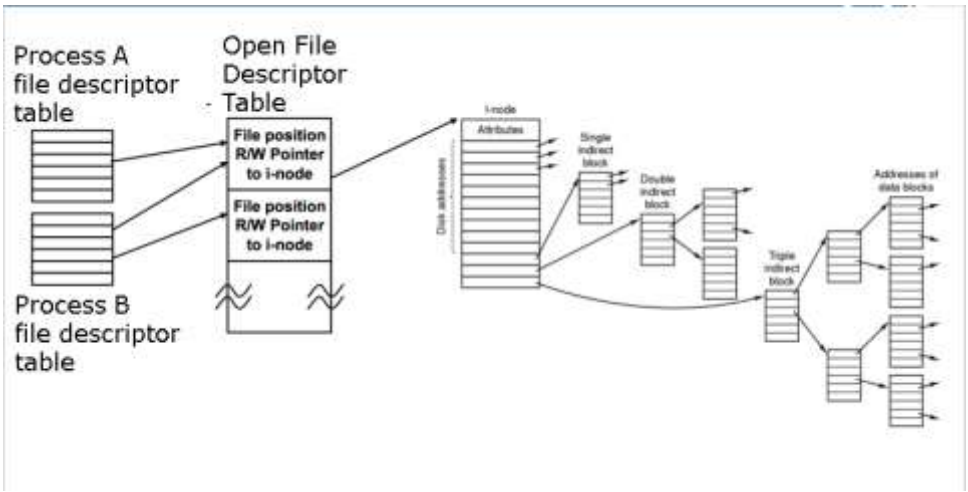
- איך בדאי לשמור- כמה שיותר רציפים אבל לא מדי
- ייצוג המיקום של הקובץ על הדיסק- iNodes and FAT
- איך יש התייחסות לקבצים בתוך הOS?

יש 3 טבלאות שנשמרות, שכל אחת מצביעה על השניה (המימוש של לינוקס, חלק מאחדות את 1 ו2):

- **פר תהליך (בתוך הPCB)**: לכל תהליך יש קישור לקבצים אליהם הוא יכול לגשת. מערכת ההפעלה עושה את האבסטרקציה של כל הקריאה מהדיסק. FD זה מצביע למיקום או למידע של הקובץ שיש לי ברשימה של הקבצים הפתוחים (הsystem wide table).
  רוב הFS דורשות שנפתח קובץ לפני שנוגעים בו. בעצם ככה מוסיפים לטבלה בPCB מידע.
  הבעיה מתחילה כשכמה יוזרים ניגשים לאותו קובץ. הסינכרון בין תהליכים נעשה ע"י IPC- inter process communication.
- **system wide table**: יש לה רשימה של כל הקבצים הפתוחים, לפי תהליך. אם אותו קובץ נפתח ע"י 2 קבצים שונים, יהיו 2 שורות שונות. זה כולל אופסט של איפה התהליך נמצא בקובץ.
- **i-node table**: מצביע לקובץ עצמו, איפה הוא נמצא על הדיסק, מצביע לinode, מתי ניגשתי אליו לאחרונה, מה הגודל של הקובץ ומונה של כמה תהליכים פתחו את הקובץ (כדי לדעת מתי אפשר להעיף את הקובץ).

stdin וstdout וstderr רק מדומים בקובץ ויש להם מספר FD קבוע ושמור, אבל בפועל הם לא באמת כאלו ואין להם inode. משתמשים בדימוי הזה כדי שהגישות יהיו זהות.

לחזור על זה ולוודא שמבינים כולל מיקומים:



מימוש ספריות, 2 גישות למימוש:

- וינדוס (כרגע)- קבצים מיוחדים שהם קבצי ספריה, בהם שומרים את כל המידע שנמצא בספריה- מקומות בדיסק, הרשאות עליהם וכו'.
- לינוקס- קובץ אחד שמכיל רשימה של שמות של קבצים, וכל אחד מהם מכיל את מצביע לinode (את המספר הייחודי שלו).

מערכת ההפעלה שומרת מעין תור שבו יש את כל הקריאות לדיסק שעוד לא קרו, ואז היא צריכה לבחור את הדבר הבא שאותו תעשה. רוב הזמן מתבזבז על הדזה של הזרוע, **נרצה למזער את הseek time**.

- FIFO- אבל אז יכול להיות שהזרוע של הדיסק תזוז לכל כיוון.
- SSTF- זו שהכי קרובה לבקשה הקודמת
  - טוב למהירות, אבל לא למניעת הרעבה
- אלגוריתם מעליות scan
  - זזים רק בכיוון אחד (עד למעלה ואז עד למטה)
  - בסוף הדיסק, משנה כיוון. או שבסוף הדיסק בוחר את הבקשה הרחוקה ביותר ומתחיל שוב.

תיזמון

מי הבקשה הבאה שאותה נרצה לשרת ולמשך כמה זמן?

preemptive scheduler- הוצאה של תהליך שפעל הרבה זמן.

המתזמן הזה CPU scheduler פועל המון, וההתייחסות לתהליכים היא כלjobs.

המבצע הוא הdispatcher שכבר דיברנו עליו.

יש מתזמן אחר midterm scheduler שמעביר בין מצב ready לready suspended ובין waiting לsuspended blocked (מהרצאה 9, כדי שלא יהיו המון page faults).

נפריד בין 2 סוגי ג'ובים:

- CPU bound- שבקושי צריכים IO, לרוב יש צורך בCPU. יש לו cpu burst ארוכים.
- IO bound- הרבה יציאות לIO, עם cpu bursts קצרים.

יש מהון סוגי מתזמנים בעולם. איך מודדים מהו מתזמן טוב?

- **throughput** (# הג'ובים שסיים)/(יחידת זמן) מספר הג'ובים שמצליח לסיים ביחידת זמן אם המערכת יציבה, זה זהה ל(# הג'ובים שהתחיל)/(יחידת זמן)
- **turnaround time** הזמן הכולל שלוקח לסיים ג'וב (הזמן שחיכה + הזמן שרץ). לא הכל בשליטת המתזמן. נקרא גם Latency.

יש עוד מדדים בהם משתמשים לפעמים

- **response time** זמן עד שהcpu burst הראשון נגמר
- **cpu utilization** כמה מתוך הזמן כולו הוא עושה דברים מועילים
- **fairness** הוגנות בין ג'ובים

הרבה פעמים המטרות האלו סותרות אחת את השניה.

כרגע אנחנו מתייחסים כאילו קונטקסט סוויץ' לא עולה לנו. סוגי מתזמנים:

- האם אנחנו יודעים את העתיד? (אפשרי בתהליכים מחזוריים) offline vs online
- גודל הג'ובים- האם ניתן לדעת מראש?
- האם מותר לו להוציא ג'ובים שלא גמרו ולהחזיר לתור?

1. FCFS- first come first serve - יש את אפקט השיירה שהרבה ג'ובים קצרים נתקעים אחרי אחד ארוך.
2. SJF- shortest job first – יודע מה יהיה בעתיד, והמטרה שלו היא למזער את הזמן הממוצע בו מחכים. הוא אופטימלי (מבחינת turnaround time וwaiting time), ולכן preemption לא יכול לעזור.

3. SRT (preemptive SJF) – אם לא כולם מגיעים בבת אחת. לוקחים את מי שזמן הסיום שלו הוא הקרוב ביותר.
   זה יכול להוביל לחוסר הוגנות ואפילו הרעבה.

איך מנבאים את העתיד? על סמך העבר. ניתן לפתוח את הנוסחה הרקורסיבית ולראות שבעצם אנחנו מסתמכים על כל העבר, אבל ככל שהוא רחוק יותר אנחנו נותנים לו פחות משקל.

4. RR round robin – בלי הנחות, ומתעסק עם הרעבה. יש קאוונטום שזה הזמן שכל תהליך יכול לרוץ, וכשהזמן נגמר הוא חוזר לסוף תור הFIFO.
   a. אין מצב של הרעבה
   b. הזמן הממוצע שבו מחכים הוא יחסית נמוך
   c. זמן התגובה יכול להיות מאוד טוב (כי הג'ובים הקטנים יגמרו מהר אם בחרנו קאוונטום טוב).

י"ד סיון ה'תש"פ, תרגול 11, איתן

תיזמון

יש כמה **אלגוריתמי תיזמון CPU**, ואנחנו מדברים רק על התהליכים שהם ready.

יש לנו כמה קריטריונים שאנחנו רוצים למקסם ולמנמם.

- First come first serve (FCFS) – זה פשוט FIFO.
  - הוגן, בלי הרעבות
  - הwaiting time יכול להיות מאוד ארוך (תהליך קצר שהגיע אחרי ארוך). אם כל התהליכים בערך באותו האורך זה כן טוב.
  - הוא non-preemptive- לא עוצר תהליך באמצע. ואם תהליך נתקע בלולאה אינסופית- כל המחשב תקוע.
- Shortest time first (SJF/SJN) – מבצע מהקצרה לארוכה
  - non-preemptive
  - ניתן להוכיח שהוא אופטימלי במצב offline- כלומר אנחנו כבר יודעים את כל המשימות
  - לא אופטימלי במצב online
  - אי אפשר להריץ כשלא באמת יודעים כמה זמן כל תהליך עומד לרוץ.
  - לא הוגן, עד כדי הרעבה (של משימה ארוכה שלעולם לא תתבצע אם ממשיכות להגיע משימות קצרות לפני שהוא מתחיל).
- Shortest remaining time first (SRTF) – כמו JSF אבל preemptive
  - אופטימלי גם באונליין
- Priority scheduling (PS) – לכל תהליך יש עדיפות, והם יקרו לפי העדיפויות שלהם (הנמוך זה העדיפות הגבוהה)
  - בחירת העדיפות היא או לפי היוזר, או שמערכת ההפעלה בוחרת- לדוגמא תהליך שהיו לו ברסטים קצרים בעבר.
  - יכול להיות preemptive או Non-preemptive.
  - בעיה של הרעבה. הפתרון הוא הזדקנות- נעלה את העדיפות גם לתהליך שלא הגענו אליו, וכך מתישהו הוא יהיה בעדיפות גבוהה.

- Round robin (RR) - כל תהליך מקבל קאוונטום (מספר במילישניות) של זמן שהוא יכול לרוץ.
- Multi-level queue – יש חלוקה לקבוצות-כל מיני תורים (למשל תור למשימות אינטראקטיביות ותור למשימות רקע), וניתן עדיפות לכל תור בזמן אחר תוך ניהול המשימות בתוך התור (יכולים להיות אלגוריתמי תיזמון שונים בכל תור).
  - יכולה להיות בעיה של הרעבה (אם התיזמון הוא שבכל שיש משהו בתור העליון נבצע אותו)
  - נוכל לעשות חלוקה זמן בין התורים (לתת קאוונטום גדול יותר לתורים עם עדיפות גבוהה יותר)
- Multi-level feedback queue – כמו הקודם, רק כשנכנסים לתור מסוים, לפני שחוזרים לסוף התור מחליטים אם לשדרג או לשנמך את התהליך לתור אחר (אם הוא יותר אינטראקטיבי, או אם הוא לא קיבל בקושי זמן).
  - נותן למשימות קצרות/בינוניות להסתיים מוקדם ולא לחכות הרבה זמן. סוג של שדרוג לRR (בלי מצב שכולם מסתיימים בסוף יחד)

בהנחת שcontext switch לוקח זמן אפסי, FCFS או RR יותר טוב? תלוי מתי. יש מקרה שבו שניהם מסיימים באותו זמן, אבל הזמן הממוצע שבו תהליך מסיים הרבה יותר גרוע בRR! זה לא מוצלח כשכל התהליכים לוקחים בערך אותו זמן, ומוצלח לחיים האמתיים.
לcontext switch יש עוד מחירים מעבר לזמן עצמו- הוא הורס את הקש.

תזמון משימות במחשבי על- חווה של המון שרתים שצריכים לשרת ענן מסוים.
לכל מחשב יש מתזמן משלו, אבל יש גם מתזמן שבחור איזו משימה לשלוח לאיזה מחשב. לא יהיה preemption כי החלפה עולה המון (העברה של מידע ברשתות תקשורת). המתזמנים הם:

- FCFS – כמו טטריס. הכי פשוט והוגן אבל קצת מוגבל.
- backfilling- אם אין מקום לראשון, כן נכניס את מי שהגיעו אחריו עד שיהיה לו מקום
  - יכול לגרום להרעבה (במקרה שיש משימה שצריכה את כל המעבד)
- EASY – משתמש בbackfilling על בסיס זמן הגעה- הזמנה של זמן בעתיד

כשמשתמש מעלה תהליך, הוא צריך לתת את כמות המעבדים שהוא צריך, והשערה של זמן שיקח לו לרוץ. אם התהליך לא הסתיים בזמן שהמשתמש נתן- הוא יהרג (זאת בעיה של המשתמש).

ב"ג סיון ה'תש"פ, הרצאה 12, דוד

תהליכים בwaiting מחכים למשהו אחר (עכבר, מקלדת) ואנחנו לא עושים עליהם תיזמון.

Round Robin

n ג'ובים וקאוונטום q- עבו ג'ובים גדולים כל ג'וב מקבל 1/n מהמעבד, ומחכה לכל היותר q(n-1) זמן. q גדול- יהיה FCFS. q קטן- המון CS והתקורה תהיה גבוהה. לכן נעדיף לבחור q שהוא קצת יותר גדול מהמשימות הקטנות שיש לעשות. q קטן עדיף בשביל ג'ובים שהם אינטראקטיביים. q גדול עדיף למערכות שבהן יש הרבה batching.
גם כשאין אף אחד בתור עושים CS – כי המתזמן צריך לרוץ בעצמו על הCPU (היחיד שיש לנו) כדי שיוכל לבצע החלטות. הוא רץ כשתהליך מוותר בעצמו על הCPU, והדרך המקובלת היא לתמוך ברמת החומרה בעזרת פסיקת שעון.

במציאות נרצה גם לתעדף לפי תורים של עדיפות גבוהה ועדיפות נמוכה, ובכל שלג'וב יש עדיפות גבוהה יותר, הוא יקבל יותר CPU ויותר מהר (ואם נמוך יכול להיות שלא יקבל בכלל).

הפתרון להרעבה הזו היא:

- **חלוקת זמן**: כל תור מקבל חלק (לא שווה) של הזמן לרוץ, שהוא נותן לג'ובים שבו (לפי תיזמון שהוא רוצה).
- **הזדקנות**: ככל שתהליך נמצא יותר זמן במערכת, העדיפות שלו עולה ("והדרת פני זקן").

יש גם תעדוף שיורד אבל הוא לא מונע הרעבה, בהתחלה כולם מקבלים את התעדוף הגבוה ביותר ואחרי כל ריצה יורדים תור.

Multilevel Feedback Queue - נצטרך להגדיר:

- מספר התורים
- אלגוריתם תזמון לכל תור
- מתי משימה עולה תור ומתי יורדת
- באיזו עדיפות מתחילים

התזמון יותר מסובך כשיש כמה CPU.

תיזמון גם עובד כשיש משימות שיש להן דד-ליין (זמן שבו היא חייבת לסיים או רצוי שבו תסיים). יש המון סוגי מתזמנים! לכן יכולים לשאול אותנו על אחד שאנחנו לא מכירים חלק מהדברים קשים למימוש, וצריך לקחת בחשבון שזה צריך להיות מאוד מהיר.

I/O

התקני קלט/פלט כוללים הרבה דברים שדרכם אפשר:

- ליצור קשר עם המשתמש. יש כאלה שרק פולטים- מקלדת, עכבר. יש כאלה שרק מקבלים פלט- מדפסת, מסך. ויש כאלה שגם וגם- כרטיס רשת.
- התקני איחסון – את רובם היום אפשר גם לקרוא וגם לכתוב (דיסק, וכאלה שרק קריאה- CD-ROM)

מה התפקיד של מערכת ההפעלה כאן?

1. להציג את ההתקנים בצורה אבסטרקטית לוגית
   a. להסתיר את פרטי המימוש
   b. לנהל שגיאות
2. לעשות פעולות במעבד בזמן שההתקן פועל
3. חלוקה של משאב בין שני תהליכים
   a. הגנה בין שני תהליכים שונים
   b. אלגוריתם תזמון בין העבודות השונות (של מדפסת לדוג')

לhardware יש גם תוכנה- מעבד קטן שעושה דברים מאוד ספציפיים ועובד במקביל למעבד של המחשב. הוא נקרא device controller/adapter ורץ על הCPU של ההתקן.

device driver רץ על הCPU של המחשב בקרנל מוד, מריץ אותה ומודיע שסיים. האינטרפייס בין הדרייב לקונטרולר הוא אינטרפייס ברמה מאוד נמוכה, דרך רגיסטרים. יש רכיב חומרה שאחראי לקשר בין הדברים.

יש שתי אופציות- העברה של הודעות לפי API מסוים וברמה של ביטים, או זיכרון משותף.

הבעיות הן איך מעבירים את המידע, ואיך מודיעים לOS ולאפליקציה שסיימתי?

1. Polling – הCPU כל הזמן בודק מתי ההתקן סיים. כותבים את ההוראה, ואז מחכים שהביט של BSY מפסיק להיות דלוק (דרך בדיקה כל הזמן).
2. פסיקות (interrupts)- הCPU מעביר את המידע, אבל פסיקה מודיעה לCPU כשהIO מסיים.
3. DMA- Direct Memory Access – ההעברה בין הבאפר לזיכרון המרכזי יכולה להיות ישירות, הCPU רק נותן את ההוראה שזה יקרה. יש אינטרפט כשמסיימים.

הבעיה היא שעדיין שניהם מתחרים על גישה לזיכרון המרכזי. זה בסדר בעיקר כי יש את הcache באמצע, שרוב הדברים של הCPU מתבצעים בו.

בתוך מערכת ההפעלה יש איחוד של הדרייברים תחת חלוקות מסויימות- כאלה שעובדים עם בלוקים, כאלה שעובדים בסטרימים, כאלה שעובדים עם פאקטים- network communication. כי יש להם הרבה דברים משותפים, וזה חוסך חלק מהמימוש. מעליהם יש אבסטרקציה לדברים לדוגמא לניהול הקבצים.

כ"ד סיון ה'תש"פ, תרגול 12, עידן

חזרה

אינטרפטים

הודעות שנשלחות למערכת ההפעלה כדי להודיע לה על אירועים שדורשים את תשומת הלב שלב ב"מיידי". יש כאלה שאפשר לדחות את הטיפול בהם, ויש שלא.

- אינטרפטים חיצוניים- נקרא גם hardware interrupt, קורים בעקבות דברים בחומרה
- אינטרפטים פנימיים- נקרא גם software interrupt / exeption, קורה בגלל התוכנית הנוכחית, לרוב שגיאה בה (למשל חלוקה בס, גישה למקום שלא שלך, הוראה שאין הרשאות לבצע, הוראה לא חוקית, page fault, trap- המימוש של syscall)
  - בpage fault מערכת ההפעלה מטפלת בו מאחורי הקלעים בלי שהתהליך מודע

בסוף הטיפול באינטרפט התוכנית חוזרת לשורה הבאה שהיא צריכה להריץ.

סיגנלים

הודעות שנשלחות כדי להודיע על אירוע כלשהי. ההודעות האלו נשלחות לתהליך. יש טיפול דיפולטיבי, ואפשר גם להגדיר טיפול ספציפי (דריסה) לחלק מהסיגנלים.

PCB- בלוק שמערכת ההפעלה שומרת לכל תהליך, שמכיל את:

- מידע של התהליך- ערכי הרגיסטרים, מצביעים לחלקי הזיכרון, מצביעים לטבלת הדפים, משתנים סביבתיים (טבלת FD לדוגמא).
- מידע של הOS- עדיפות, הרשאות גישה, מצב ריצה.

Threads

הרצה במקביל (באמת אם יש כמה מעבדים), יותר זולה מאשר תהליכים מבחינת תקורה של מיקום בזיכרון וכו', זמן טעינה גדול, שיתוף מידע יקר. threadים משתפים חלקים בזיכרון (חוץ מהסטאק).

- kernel level threads – תהליכים "קלים" יותר. מי שמנהל את התיזמון זה מערכת ההפעלה.
  - מערכת ההפעלה תכיל TCB לכל טרד, מתוך הPCB.

- user level threads – מנוהל על ידי המשתמש. הסטאק של כל טרד הוא על ההיפ. התיזמון נבחר ע"י המשתמש. חוסכים הרבה תקורה של מעבר למערכת ההפעלה שתבחר את הטרד הבא שירוץ. כאן לא מנצלים את הmulticure שזה חסרון.
  - אם אלגוריתם התזמון הוא לא preemptive – לא צריך לדאוג למניעה הדדית

Critical section problem

יש n תהליכים, בלי הנחה על מה הם עושים חוץ משימוש במשאב משותף.

הגדרנו 5 קריטריונים להאם אלגוריתם למניעה הדדית הוא מוצלח או לא. יש פתרונות:

- ברמת החומרה
  - text&set- לא מונע הרעבה
- ברמת מערכת ההפעלה
  - סמפור – הכללה של מיוטקס לכמה תהליכים (שומר כמה תהליכים יכולים להכנס)
  - מיוטקס
- ברמת התהליך
  - האלגוריתם של פיטרסון - טוב רק ל2 תהליכים

Deadlocks

יכול לקרות אם 4 תנאים קורים במקביל:

1. מניעה הדדית – רק תהליך אחד יכול לגשת למשאב בזמן מסוים
2. hold and wait – תהליך שמחזיק משאב מסוים יכול לבקש עוד משאב
3. no preemption - אי אפשר להפקיע משאב מתהליך
4. circular wait – של לפחות 2 או יותר

דרכים להתמודד עם דדלוק (אין פתרון מושלם לזה):

- להתעלם
- זיהוי ופתרון – תריץ סריקה פעם בכמה זמן, ואחרי הזיהוי תנסה להשתקם
- avoidance – בדיקה לפני כל הקצאה
- prevention – לדאוג שתמיד אחד התנאים לא יתקיים

Scheduling

אם היינו נותנים לכל תהליך לרוץ מההתחלה ועד הסוף, לא נוכל להגיב לדברים שקורים ברקע ונאבד מהביצועים, הניצולת נמוכה ואי אפשר להריץ משימות במקביל.

יש לנו כמה אלגוריתמים לבחירה של את מי להריץ. יש כמה קריטריונים לתזמון:

- CPU utilization – מקסימלי
- throughput – מקסימלי
- waiting time – מינימלי
- turnaround time – מינימלי

Caching

יש כמה רמות של קשים. יש כל מיני אלגוריתמים לניהול של קש.

ניהול הזיכרון- יש הבדל בין כתובת וירטואלית לכתובת פיזית.

לחזור החל מ33

תקשורת- דרך:

- זיכרון משותף
- sockets- TCP אמין, UDP ולא אמין.

- pipes – מעין צינור שמחבר בין 2 "קבצים"-משאבים. פונקציות: להבין מה עושה
  - pipe - (syscall) שיוצרת שני משאבים, אחד לקריאה ואחד לכתיבה
  - dup, dup2 – משכפלות FD, כך שיש שני FD שמצביעים לאותו קובץ

File system

lseek מזיזה את הראש הקורא. לסיים

hard links – שם של הקובץ (כל המסלול אליו). אפשרי לעשות שלאותו קובץ יש כמה שמות (עם הפונקציה link, וניתן גם לנתק עם unlink), ואז סופרים כמה שמות יש לקובץ- והוא ימחק רק כשיש אליו 0 מצביעים.

shortcut בווינדוס זה soft link.

לעבור על שאלות החזרה

א' תמוז ה'תש"פ, הרצאה 13, דוד

I\O

גם הדרייברים מאוגדים לקבוצות של סוגים של חומרה שמתנהגים אותו דבר מבחינת IO: בלוקים, סטרימים, נטוורק. למערכת ההפעלה יש רכיב שמנהלת את כל הרכיבים מאותו סוג.

לדוגמא file system משמש בתור האינטרפייס של הOS בשביל דברים שהם בבלוקים, אבל הרבה פעמים עושים לו אוברלואד כדי להתקשר מול דברים שהם בסטרימים (0 ו1 בשביל stdin ו stdout).

מה device-independent techniques יכול לכלול (פירוט במצגת):

- buffering/caching
- error handling טיפול בשגיאות שקורות במעבר של המידע (סכימה של כל ה1 ובדיקה האם המספר זוגי או אי זוגי)
- device scheduling/sharing – לדוגמא למנמם את ההזזה של הזרוע על הדיסק

Virtualization

לוקחים משהו מדומה ועושים עליו פעולות ממשיות. לדוגמא:

- תהליך שרץ ומרגיש כאילו הCPU רק שלו
- חלוקה של הדיסק לחלקים, שיוצרת תחושה שיש כמה דיסקים כשבפועל יש רק אחד
- VPN גלישה ברשת פרטית, אבל בעצם עוברת דרך רשת ציבורית

אותנו מעניין virtual machine- מכונה שלמה (שכוללת את כל החלקים של המכונה) נפרדת, שחושבת שהיא מכונה אמיתית. האפליקציה שרצה בVM לא אמורה להיות מודעת לזה שהיא רצה בסביבה וירטואלית (מבחינת הפקודות והזיכרון). מבחינת ביצועים הרבה פעמים אפשר לנחש שרצים בסביבה וירטואלית ולא אמיתית. אם יש כמה VM שרצים על אותו סרבר, הם לא יודעים שהם רצים במקביל ולא יכולים להעביר מידע ביניהם. כל זה צריך לקרות במהירות שקרובה למהירות האמיתית.

האתגר הוא ליצור פלטפורמה שלוקחת את החומרה ונותנת להרבה מופעים של OS לרוץ עליה.

לשכבה שיוצרת את המעבר מהVM לחומרה קוראים Virtualization layer /platform /VMM /Hypervisor שעושה את המעבר בין החומרה המדומה לחומרה האמיתית.

צריכה למפות בין הפעולות שקורות בחומרה הוירטואלית לאמיתית (קריאה מהמקלדת), ולעשות isolation בין VM שונים שרצים על אותה מכונה אמיתית.

<u>למה צריך את זה?</u>

- הרצה בענן
  - שימוש בחומרה שלא משתמשים בה
  - גמישות של כמה דברים אנחנו רוצים לשים על אותו סרבר (כדי לחסוך את מספר הסרברים)
  - בידוד חזק בין VM שונים
  - חוסך באנרגיה של הפעלה של סרברים שונים (חוסך בחשמל ובחימום של הסרברים)
- שיחזור מאסון
- תמיד זמין להשתמש בזה
- בדיקה בלי תקיעה של כל המחשב אלא רק של הVM

כשמריצים ככה על ענן, סומכים על זה שמידע לא יזלוג ושאם יקרה משהו לחומר האמיתית- נעבור לאחת אחרת.

כל זה בנוי על workload multiplexing – אנחנו לא משלמים פעמיים את המקסימום אלא בגלל שלרוב הזמן שהVMים צריכים את מקסימום המשאבים זה לא בדיוק באותו רגע, אנחנו חוסכים הרבה משאבים.

השימוש בVM encapsulation- איך זה נראה מצד המשתמש- זה בעצם הכל תוכנה, ולכן אפשר לשמור לקובץ את כל המצב שבו התוכנה נמצאת. זה נקרא הimage של הVM, ששומר את המצב של כל המכונה, ואפשר לקחת והריץ אותו על חומרה אחרת. אפשר לשמור את הצילום הזה פעם בכמה זמן ואז אם החומרה קורסת ניתן לשחזר, או להעביר בקלות בין חומרות. אפשר להעלות VM ואז לשמור את המצב הקיים, ואז איתחול שלו יהיה מאוד מהיר.

סוגים של hypervisors-

1. שכבה של וירטואליזציה במקום הOS
2. הרצה בתוך מערכת הפעלה קיימת (כמו האובונטו שלנו), ואז השכבה של hypervisor היא מעל הOS ועליה אפשר להריץ VM רגילות – רץ בתור אפליקציה בתוך הOS המארח
3. הרצה מעל OS קיים, שכל קונטיינר עושה וירטואליזציה (דומה לתהליכים, אבל קונטיינר הוא אוסף של תהליכים שיכול גם להתקים דברים- שקונטיינר אחר לא מודע אליהם) למערכת ההפעלה.

שיטות של וירטואליזציה:

1. trap and emulate – רץ בuser mode כדי להגן על החומרה מפני מישהו זדוני.
   הVM חושב שהוא רץ בקרנל מוד- קוראים לזה virtual kernel mode. השליטה אחרי גישה כזו חוזרת לhypervisor והוא מתרגם את הפעולה למה שצריך לעשות ומחזיר אותה (הוא כן עובד בkernel mode).
   system calls בתוך הguest OS נשארים בתוכו.
2. binary translation – את כל הפעולות של הvirtual kernel mode מעבירים דרך הhypervisor שבודקת אם בלוקים של פקודות הם בסדר, ואם לא אז מתרגמים תוך כדי לקוד בינארי בלי גישות previleged.

3. paravirtualization – נותן למערכת ההפעלה של הVM לדעת שהוא רץ בסביבה וירטואלית, וקוד המקור שלו משתנה כדי לתמוך בדברים האלו.
4. hardware assistance – תמיכה ברמת החומרה, הוספה של עוד מצב ריצה- root ושל עוד פעולות (פתיחה וסגירה של מכונה וירטואלית) ועוד מבני נתונים שגורמים למעבד להיות מודע לקיומה.

אחד הדברים שהמכונה הוירטואלית צריכה לעשות זה כתיבה לזיכרון- כולל הpage table, ואז צריך להבין איך שומרים על קונסיסטנטיות, איך מעדכנים את הTLB ועוד.
אם שי יכולות לעשות paravirtualization יש הרבה שיפורים שאפשר לעשות.
מאז 2008 (אופציה 4) יש אפשרות לעשות nested paging.

שיטות לIO virtualization:
1. emulated (כמו 1 בקודם)
2. split- יש רכיב בquest OS שמדבר עם הdevice driver של הhypervisor (כמו 3 בקודם, ברמת הדרייבר) עם אינטרפייס פשוט ביניהם
3. pass-through – נותנים לVM לדבר ישירות עם החומרה. בעיקר משתמשים לרשתות תקשורת שצריכות מעבר מאוד מהיר. צריך תמיכה של התוכנה לוודא בידוד ובטיחות

<u>קונטיינרים</u>
הרבה פעמים מספיק לעשות וירטואליזציה של מערכת ההפעלה (ולא של כל המכונה). לכל אחד מהקונטיינרים יש את כל הרכיבים משלו והוא לא יודע על קונטיינרים אחרים שרצים על אותה מערכת הפעלה. זה משהו בין תהליכים לVM.

VM	**קונטיינר**
מכילה בתוכה מערכת הפעלה, תמונה שלה היא מאוד גדולה. ברוב המקרים זה לא נחוץ, והייתה מספיקה רק סביבה.	סביבה שרצה בתוך מערכת הפעלה ולכן משתמשים בכל מה שהיא נותנת כדי לדבר עם ההתקנים.
בידוד ובטיחות גדולים יותר	בידוד ובטיחות פחות טובים. החסרון הכי גדול
יכולים לרוץ עם מערכות הפעלה שונות על אותו סרבר	צריכים להיות עם מערכת הפעלה זהה. גם חסרון
וירטואליזציה של הhardware	וירטואליזציה של מערכת ההפעלה (קל יותר)
	קטנים בהרבה (אפילו פי 100)
	להקים קונטיינר זה הרבה יותר קל (כמעט כמו להקים תהליך)
	הכל קורה מהר יותר- ולכן חוסכים כסף

תוכן עניינים