

מערכות הפעלה | 67808

הרצאות | פרופ' דוד חי ופרופ' דרור פיטלסון

כתביה | נמרוד רק

תשפ"ב סמסטר ב'

תוכן העניינים

I מבוא, אבסטרקציה וירטואלייזציה
6 הרצאה
6 אבסטרקציה
6 וירטואלייזציה
8 סקירת הקורס
9 המשך וירטואלייזציה
10 תרגול
II מצב קרnl ו互動ראפטים
11 הרצאה
12 מצב גרעין
14 התקני IO
14 אינטראפטים
15 תרגול
III תהליכיים וחוטים
17 הרצאה
17 תהליכיים והkontext שלם
19 Multiprocessing
20 חוטים
21 חוטים ברמת הkrnl והמשתמש
23 תרגול
IV סכון
27 הרצאה
27 דוגמאות לבעיות סינכרון
29 המודל של Djikstra ודרישות לאlg' מניעה הדדית
30 פתרונות למניעת הדדית
32 אלג' המאפייה של למפורט
34 פעולות אוטומיות מורכבות
35 תרגול

41	V סמאפורים ומוניטוריים
41	הרצאה
41	סמאפור
42	מגבליות הסמאפור
43	בעית הפילוסופים האוכלים
44	תנאי קופמן הכרחיים לדד-лок
44	יצרן-צרכן
45	מוניטוריים
46	תרגול
54	VI תזמון תהליכיים
54	הרצאה
54	מטרות התזמון
55	אלגוריתמים אופלין
55	אלגוריתמים אונליין
57	VII תזמון לעומק
57	הרצאה
58	טור פידבק רב שלבי
58	הרעה בתור פידבק רב שלבי
59	הערכת ביצועי מותזנים
60	ניתוח של מודל פשוטני
61	מערכות פתוחות וסגורות
62	توزמון מתקדם
63	תרגול
66	VIII ניהול זיכרון
66	הרצאה
66	המטמוני
68	סגמנטציה
69	פרוגמנטציה ופתרונוותיה
71	Paging
71	Paging
71	IX דפדף
71	הרצאה
71	גודל הדף האופטימלי
72	דפדף לפי דרישת
74	קורבנות ואלגוריתמי גירוש
76	תרגול

78	X דפדף לעומק
78	הרצאה
78	ביצועי דפדף
79	локאליות בזמן
80	Thrashing-ו Swapping
81	טבלאות דפים אלטרנטיביות
82	הננה ושיתוף
83	קבצים
83	תרגול
88	XI מערכות קבצים
88	הרצאה
88	שמות במערכת קבצים
89	שימוש של תיקיות
90	הננה על קבצים והרשאות
91	שימוש מערכות הקבצים בדיסק
92	גישה לבלוקים השبيיכים לקובץ
93	תרגול
97	XII וירטואלייזציה
97	הרצאה
97	פרטים נוספים על מערכות קבצים
99	ווירטואלייזציה
100	היסטוריה ותורנות וירטואלייזציה
100	שימושים של וירטואלייזציה
101	амצעי וירטואלייזציה
102	Containers
103	תרגול
106	I/O XIII
106	הרצאה
106	השלמות של וירטואלייזציה
107	התקני I/O מבחרית מערכות הפעלה
107	דרייברים וקונטולרים
108	USB-ו Port Parallel
109	שיטות מעבר מידע
110	שכבות ביןיהם ותוספות
111	תרגול

113	XIV אבטחה
113	הרצאה
114	פריצת סיסמאות
115	הרשאות
115	מפגעי אבטחה
117	תרגול
122	XV חזרה
122	תרגול

שבוע ॥ | מבוא, אבסטרקציה ווירטואלייזציה

הרצאה

מהו מחשב? מחשב מורכב מחומרה שעלייה מרכיבים אפליקטיבים (תהליכיים, לעתיד) באמצעות מערכת הפעלה. משתמשים ונתונים משתמשים בה, אך אינם חלק ממערכת מחשב.

בין האפליקציות לחומרה יש מערכת הפעלה, והיא חוצצת (לרוב) ביןיהם. מערכת ההפעלה לא תמיד רצאה, אלא רק כשהיא מותבקשת על ידי האפליקציה (לדוגמה הצגה למסך). על מערכת ההפעלה (לרוב) רצות הרבה מאוד אפליקציות בו-זמנית.

החיצזה זו מורכבת משני ממשקים:

- ארכיטקטורה (בין החומרה למערכת הפעלה): אילו פקודות המעבד מימוש וכייד קוראים להן.
- קריאות מערכת (בין האפליקציות למערכת הפעלה): פעולות ושירותים שניינימם לאפליקציה.

מערכת הפעלהקובעת איזו אפליקציה מקבלת את כוח החישוב של המעבד וכשהיא רצאה (קובעת איזו אפליקציה תרצו לדוגמה) אף אפליקציה לא רצאה ולהפוך. לאחר זמן מה שאפליקציה רצאה, קורת פסיקה של שעון המעבד (ש망תק 1000 ~ פעמים בשנייה) ומערכת הפעלה עוברת להרצת אפליקציה אחרת.

מערכת הפעלה מספקת סביבה נוחה ובטוחה לאפליקציות (באמצעות וירטואלייזציה ו-abs-traktsia). מערכת הפעלה היא ריאקטיבית, כלומר היא מתחילה שימושו יקרה. בנוסף היא תמיד שם, לעולם לא מסיימת את פעולתה ונמצאת בלאה אינסופית.

אם לא הייתה לנו מערכת הפעלה, היינו צריכים לשים את הקוד במקום ספציפי כדי שכאשר המחשב נדלק הוא יתחל להריץ את הקוד הנוכחי, להתמודד עם זה שאינו קבצים בכלל, לדבר ישירות עם הכביר/מקלדת וכל זה באופן לא בטוח ובליל מיקובל של פעולות.

על מחשב אחד ניתן להריץ יותר מערכות הפעלה אחת (ווירטואלייזציה) ולהפוך (ארכיטקטורות רבות-מעבדים) אך לא נ逋וק בכך בקורס הזה.

אבסטרקציה

מערכת הפעלה מציגה לאפליקציות מכונה אבסטרקטית, שמכילה פעולות שלא קיימות בחומרה אך מtabססות עליה. זה מקל על כתיבה של קוד והוא מפחית את המרכיב לחזקה הרבה יותר מאשר שימוש סטמי בחומרה. עבור דברים שדורשים עילויות יתרה, לעיתים כן כדאי לעבוד עם חומרה ולהיפטר מהאבסטרקציה.

חומרה	מערכות הפעלה
שמירת מידע בקבצים בעלי שם	שמירה בבלוקי מידע
גודל קבוע	גודל משתנה ככל שנרצה
קריאה לפי מקום על דיסק (משטח, מחלקה, מסלול)	קריאה לזמן לזמן ויחס
-	מבנה נתונים היררכי

טבלה 1 : קבצים דוגמה לאבסטרקציה

מערכת הפעלה מגבילה את "שדה הראייה" של כל אפליקציה למוכנה אבסטרקטית שמכילה רק את מה שהאפליקציה צריכה לראות, ומוכנה זו יכולה להיות שונות בין אפליקציות (מקום זיכרנו שונים).

מכונה וירטואלית	חומרה אמיתית
פקודות מכונה (עם ובלי הרשות)	פקודות מכונה ללא הרשות
4 ג'יגה זיכרון רציף	4 ג'יגה זיכרון רציף (כמה ג'יגה) ומטען
קבצים בעלי שם	דיסק מסודר לפי בלוקים

טבלה 2 : וירטואליזציה של מערכת הפעלה

וירטואליזציה

לנתק את החומרה מהמגבליות הפיזיות שלה - יצירת אשלייה. כל תוכנה חשבת שהמעבד הוא שלה, אף"פ שזה לא נכון ולא תמיד נrisk את התוכנה של האפליקציה הזאת.

מבחינת זיכרון כדוגמה לוירטואליזציה, האפליקציה חשבת שיש לה 4 ג'יגה (במערכות 32 ביט) של זיכרון רציף (שנitinן לקרואו לו עם פוינטור בסיס והיסט) ושלה יש זיכרון יהודי, מערכת הפעלה תומכת באשליה הזו ומקרה זיכרון בעת הצורך ואילו לחומרה יש רק 2 ג'יגה. למעשה לאפליקציה יש הרגשה שיש לה הרבה יותר זיכרון משיש לה באמות וזוו אחריות מערכת הפעלה לספק את הדמיון הזה לה ולשלל האפליקציות האחרות שחוובות כך גם כן.

ההבדל בין אבסטרקציה לוירטואליזציה לא תמיד ברור אבל ככל אבסטרקציה משפרת ווירטואליזציה יוצרת עוד.

מערכת הפעלה ממונה על הקצאת משאים (זיכרון, זמן מעבד) ונמדד את איכותה בהקשר זה באמצעות התחשבות ביעילות, ביצועים והוגנות.

טרמינולוגיה ביצועים

1. זמן תגובה/השהייה | Latency/Response time - הזמן שלקח לבצע פעולה/להגיב (שניות).
2. תפוקה | Throughput - הקצב שמתבצע עובדה (אחד חלקי שנייה, הרץ).
3. ניצולות | Utilization - חלקו הזמן שהמחשבעובד ולא נח.
4. תקורה | Overhead - המשאים המונוצלים באופן מושני/נוסך כדי לעשות משהו, לדוגמה ריצת מערכת הפעלה.

סכום הניצולות (שאפליקציה בקשה באופן ישיר) + תקורה + זמן מנוחה מסתכם ל-100% מהזמן.

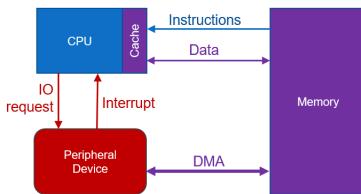
Multiprogramming

עד לבוא מערכות הפעלה, תוכניות היו רצوت אחת אחרי השניה. מערכת הפעלה חדשה ועתה מכילה Job, בה נמצאות כל העבודות שהמעבד צריך לבצע. מערכת הפעלה דואגת לחוiotת משתמש יותר נעימה כי היא מריצה במקביל כמה דברים (לא באותם במקביל, אלא מעבר מהיר בין אחד לשני). דוגמה קלאסית לכך היא Spooling, לפיה בזמן ריצה של רכיב O/I באפליקציה אחת ניתן לאפליקציה אחרת לרוץ.

הנחות התאוריתיה

- המעבד תמיד שם וזמן מנוחה שלו הוא ביוזבו.

- המעבד מהיר בהרבה מה-I/O.
- הזיכרון מספיק גדול כדי שירצו הרבה תוכנות.
- הגישה לזיכרון היא ישירה (DMA), כלומר יש לנו מטמון שצמוד למעבד שהוא מהיר ומכל מידע חשוב וכל מידע אחר ניתן לשלוות לזיכרון להביא לנו ולעשות דברים אחרים בזמן זהה (עד שיקרא לנו חזרה עם interrupt).



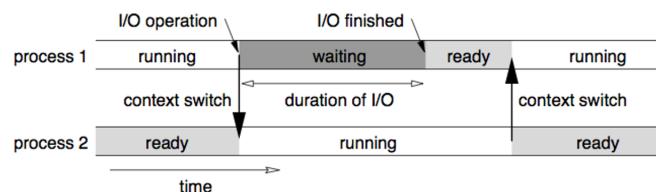
איור 1 : תרשימים שמדגים את השימוש ב-DMA

- יש יותר מדבר אחד לעשות בכל רגע נתון.

דרישות מערכת הפעלה

- תקשורת פרימיטיבית עם O/I.

- מתוזמן, שיקבע איזו עבודה כרגע רצה על המעבד.



איור 2 : הדוגמה של multiprogramming

בדוגמה הנ"ל מערכת הפעלה אחראית על המעבר בין התהליכים, בדיקה מתי ה-I/O מסייםים לעבוד, שימוש ב-I/O ועוד.

- spooler שינהל O/I יותר איטי.

שימושים של מערכת הפעלה

1. הרצת תוכנה.

2. ניהול תהליכיים (הריגת תוכנות אם הן נתקעו לדוגמה).

3. ניהול זיכרון (הקצאת זיכרון).

4. מערכת קבצים (שמירת היררכיה).

5. רשות (תקורת אינטרנט).

6. אבטחה ובטיחות (הגנה מפני פרישות).

7. הקצאת משאבים ומעקב אחריהם.

8. תקשורת עם O/I.

9. תקשורת עם המסתמש באמצעות Command Interpreter או Shell וכו'.

בימינו מערכות הפעלה מורכבות הרבה יותר, בין אם בключиים ארכיטקטואליים כגון מערכות מרובות מעבדים ובין אם קשיים שנובעים משיינוי הסביבה, בין אם זה בשירותי ענן ובין אם זה מערכות מובייל.

בخمسים השנה האחרונות עלתה אקספוננציאלית מספר הטרנזיסטורים במעבד, כגדלו עומד כתע על כמה אנומטרים (אחד חלק 10,000 שערה). קיבל השעון עלה אקספוננציאלית גם הוא עד 2005 עד שנקבע שהחומר שהם יוצרים חמור מדי ומماז הוא נשאר קבוע, ואיתו גם כמות החשמל שנדרשת לשימוש במעבד. מה שהחל לעלות אחרי 2005 הוא מספר הליבוט (המעבדים) בתוך כל מעבד, ומספר זה חסום על ידי 10 כרגע בטלפוןינו ומונה עשרות במעבדים מתקדמים.

בגלל שהארגון מגביל את מהירות השעון, הפתרון הוא מיקבול, וזה הסיבה שעברו לרכיבי מעבדים.

דוגמאות לוירטואלייזציה

1. פרטישנים (partitions) בדיסק הם חלוקות לוגיות של כונן קשיח שמתחנגים כמו דיסקים שונים אך אינם באמת כאלה.

2. זיכרון וירטואלי - הזיכרון הרציף המוגבל שיש לכל אפליקציה.

3. VPN, שמאפשר למשתמש להתחבר לאינטרנט ממוקומו האמיתי.

אפשר לעשות וירטואלייזציה גם למערכת הפעלה, כולם לשים hypervisor בין החומרה למערכות הפעלה כך שכל מערכת הפעלה תחשוב שהיא היחידה ושיש לה זיכרון כלשהו שלא בהכרח חייב להיות קיים במציאות ואפשר להוסף עוד שכבות וירטואלייזציה כאלה נפשנו (באמצעות VM שרך על מערכת הפעלה שרצה בתוך VM וכו') ויכולות להיות דרגות שונות של וירטואלייזציה באותה המכונה (אפליקציה אחת כן ב-VM ואחרת לא). יש גם container שזו אריזה של אפליקציה שהיא לא בדיקת המערכת הפעלה עצמה.

יתרונות של וירטואלייזציה

1. גיוון - כמה מערכות הפעלה על אותה מכונה.

2. סנאפושוט - להקפי את המערכת המצביע מסויים ולהשתמש בו זמן אחר.

3. הגירה - אפשר להעביר בקלות מערכת מאמצעי פיזי אחד לאחר.

4. בידוד - כל מכונה וירטואלית מופרד (כמעט) לשלוטין מהאחרות וזה מספק אבטחה.

עם כל היתרונות הנ"ל, באה גם תקורה ומשם יורדת הייעילות.

בענין וירטואלייזציה מאוד נוחה מבחינתי scalability, אבטחה, גמישות וכו' וכל זה בזול מאד.

בטלפון המכוב מאד שונה מבחינת ההנחות בעיקר כי הוא עובד על סוללה ולא חשמל ולכן במקום להשתמש במעבד תמיד נרצה להשתמש בו כמו שפהות, יש הרבה פחות זיכרון וקשה יותר לעשות יותר מדבר אחד בו בזמן.

מעבר לכך יש עוד הרבה מערכות אחרות שלחן צרכים ודרישות שונות וספקים שירותים שונים.

תרגול

סקרנו את הנהלים, שימוש ב-gdb,strace, valgrind והתרגיל.

תהליכיים שנלמד בקורס

1. מה קורה כשהוחצים על מקש במקלדת (דרייבר של המקלדת נקרא וכו')?

2. מה קורה כשהוכטבים או קוראים מקובץ (אבסטרקציה של מערכת הפעלה)?

3. אפ"ע שקריאת קובץ היא איטית מאוד, המחשב מגיב לפועלות שלנו, כיצד זה קורה (מולטי-פרוגרמים) ?

4. איך אפשר להשתמש בתוכנות שדורשות יותר זיכרון משמהחשב יכול לבצע?

5. מהו segmentation fault (גישה לזכרון לא קיים במכונה הווירטואלית)?

6. איך אפשר להריץ אפליקציות רבות ביבוט?

איך מנעים מהתנגשות בין פעולות של ליבות (סינכרון)?

7. איך אפשר להריץ יותר מערכות הפעלה אחת על מחשב (ווירטואלייזציה)?

8. איך מחשבים מתקשרים אחד עם השני?

רегистרים (אוגרים) מיוחדים

1. Program Counter - PC . שעוקב אחר אינדקס הפעולה שמורצת כרגע.

2. Stack Pointer - SP . שעוקב אחר ראש המחסנית.

3. Instruction Register - IR . עוקב אחר הזיכרון שלו אנחנו מצביים לשם שימוש בו.

סוגי פעולות המעבד

1. עיבוד מידע : טעינת בית לרגיסטר, כתיבה לזכרון.

2. פעולות אРИתמטיות : כפל, חיבור, פעולות "או" ביטית.

3. בקרת זרימה : קפיצה מותנת, לא מותנת.

מחוזר החיים של פקודה במעבד

1. בקשת פקודה.

2. פענוח פקודה ובקשה לאוגרים הנדרשים.

3. פעולות הרכיב האРИתמי בתוך המעבד.

4. קריאה וכתיבה לזכרון (sw, load, store שם load בהתקאה).

5. כתיבה חוזרת לאוגרים.

דוגמה הפקודה (sw \$1, 32 \$2) מבצעת $Reg[1] = M[Reg[2] + 32]$.

הערה גם אם יש לנו מעבד אחד, אין מנעה שהיה פקודות שונות בשלבים שונים במחוזר החיים וכך נשים שיפור קטן ביעילות.

היררכיית הזיכרון במחשב

1. זיכרון אוגרים : מאוד מהיר, עם פחות מ-1KB מקום, מספיק לחישובים זמינים.

2. זיכרון מטמון : קצר פחות מהיר, עם פי כמה וכמה זיכרון מהօגרים (עשרות KB) ומספיק לגישה מהירה לזכרון, נחשב בתוך המעבד.

3. זיכרון RAM : איטי ביחס למעבד, עם הרבה מאוד מקום אך לא מספיק לשימוש סדייר בו ולכך הומצא המטמון.

4. דיסק : הרבה מאוד מקום, מאוד איטי, המידע אף פעם לא ימתק בנגדוד לכל השאר.

על המחשב אפשר לא רק להריץ אפליקציות, אלא להוסיף שכבת וירטוואלייזציה נוספת מה שמערכת הפעלה כבר מספקת ועליה להתקין מערכות הפעלה נוספות שלא ידעו בכלל שיש משוחה מתחנן. הבועה במערכות כאשר היא שיש קורה רבה וחן דורשות הרבה משאבים. לשם כך הומצאו docker-ים, שעושים וירטוואלייזציה ברמת מערכת הפעלה. כמובן, היא לא מדמה חומרה חדשה אלא רק מערכת הפעלה חדשה, וכן יש עדין הפרדה מלאה בין האפליקציות והגישה שלחן לחומרה.

שבוע III | מצב קernal ואינטראפטים

הרצאה

סוגי פעולות

- **פעולות לא מוחסות** : פעולות שלא חשוב לנו לבקר את ההתנהגות שלהם אצל האפליקציות, לדוגמה חיבור, כפל, קפיצה וכיוצא"ב.
- **פעולות מוחסנות** : פעולות שרק מערכת הפעלה יכולה להשתמש בהן. לדוגמה קריאה לקבצים והגבלת הגישה של משתמשים שלא אמורים לגשת לבלוקים מסוימים מגלשת אליהם.

מצבי פעולה

- **מצב גרעין** : אפשר להריץ כל פעולה שהיא, מערכת הפעלה רצה במצב זה.
- **מצב משתמש** : ניתן להריץ רק פעולות מוחסנות.

מימוש מצב גרעין

יש במעבד אוגר ששמו PSW שמכיל בית מצב ואם הבית על מצב משתמש לא תינוק אפשרות להריצת פעולות מוחסנות. במעבדים מודרניים יש יותר משני מצבים ולכן יותר מבית אחד. כדי להיכנס במצב קرنל צריך לקרוא ל-trap שעשו שני דברים בו בזמן (אוטומית) :

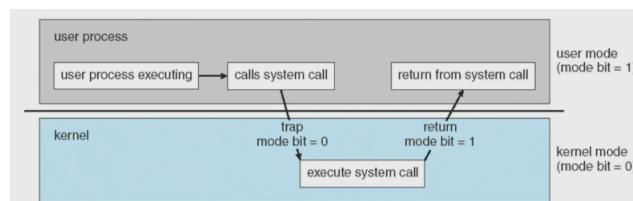
1. לשנות את בית המצב ב-PSW במצב קرنל.

2. לטען ל-PC כתובת של פ' OS.

trap היה פ' לא מוחסנת כי שגוראים לה בהכרח צריכים עדין להיות במצב משתמש.

מתי נכנסים למצב גרעין

- **כניסה במצב קرنל בקריאהsyscall** : כדי להשתמש בשירות של מערכת הפעלה צריך להשתמש ב-syscall ופקודות אלה מהוות את ה-API של ה-OS. המשמש קורא לה כמו פ' רגילה (יחסית מיוחדת אمنה) והפ' רצה במצב קرنל. בלי syscall אי אפשר לשנות רכיבים פנימיים של ה-OS כדי לשומר בטיחות המערכת. בנוסף, לפני שפועלים על הפרמטרים שהתקבלו ל-trap, תקינותם נבדקת כדי לא לבצע פעולות מסוכנות.
- בקראת syscall קוראים ל-trap, מרייצים את הפעולה הנדרשת, יוצאים במצב קرنל וחוזרים מה-syscall.



איור 3 : הדוגמה של קריאה ל-syscall

דוגמה נרצה לפתח קובץ. קוראים לפ' ספריה ששם open (פ' בספריית מערכת הפעלה) שמעבירה את הפקטרים לאוגרים ושומרת את הקוד של ה-syscall 2 (במקרה הזה). לאחר מכן, מבצעים את ה-trap ואז ה-OS מוצא את הפעולה הרלוונטיית (פתיחת קובץ במקרה זה) בהסתמך על switch-case גדול (כבר במצב קרנל). לאחר ביצוע הפעולה נחזיר ל-point entry שקרה לפעה, נחזיר במצב קרנל למשתמש ונחזיר לבסוף לפ' המקורית שקרה לנו.

הערה כל הפעולה זו מאוד יקרה.

דוגמאות ל-Syscalls

- ניהול קבצים.
 - תקשורת (אינטרנט, פרוטוקולים אחרים).
 - ניהול תהליכי (להרג, לייצור, לשנות).
 - ניהול התקנים (מדפסות, עכברים).
- **כניסה מצב קרנל בהתקלות ב-Exception**: לעיתים ה-CPU לא יכול לבצע את הפקודה שהוא קיבל, מצב זה הוא חריג ולכן נקרא exception.

דוגמה חלוקה באפס היא לא פעולה שאפשר לבצע.

דוגמה פעולה לא קיימת (בנאנד היו 4 פקודות אבל 64 אפשרויות - הרבה אפשרויות לקודים שלא קיימים) או לא חוקית (פעולה מיוחסת במצב משתמש).

בגלל שזה לרוב נובע מבאג בתוכנה, החומרה מעבירה את האחוריות לתוכנה - למערכת הפעלה. בכלל שלרוב אין מה לעשות עם שגיאה בסגנון זה (למעט בשגיאות ניהול זיכרון), מערכת הפעלה תחרוג את התהיליך באמצעות שליחת אות.

- **כניסה למצב קרנל באמצעות Interrupt** (ראו המשך).

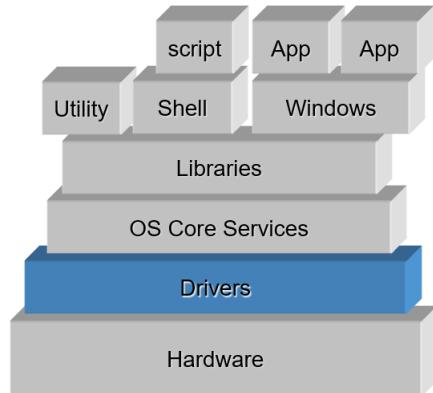
הערה לא תמיד ברור מה חלק ממכלול הפעלה ומה לא, לדוגמה האם דפדן הוא חלק ממכלול הפעלה? האם תמיינה בחלוקת והזזה שליהם היא חלק ממכלול הפעלה? לא תמיד, לעיתים זו אפליקציה בפני עצמה.

מערכות הפעלה מודרניות באות עם שירותים למשתמש שרצים כמו במצב משתמש, אך אינם אפליקציות רגילות.

דוגמה shell או echo, שיכולים לקרוא ל-syscall-ים אבל אין רצות במצב קרנל באופן ישיר.

מתוחת לאפליקציות ולשירותים האלה של ה-OS, יש את סדרות הקוד של ה-OS (ניהול זיכרון, תהילכים וכיוצא ב') שהם רוב מה שאנו לומדים בקורס זה.

מתוחת יש דרייברים, שהם סקריפטים שמתחמשים עם התקנים שונים (עכבר, מקלחת, דיסק) או מנהלים חלקים במכונה ומתוחת לכל זה מגעה החומרה.



איור 4 : הדגמה של שכבות ה-OS

בגלל שדרייברים הם חלק ממכלול הפעלה והם יכולים לגשת לכל רכיב של ה-OS, הם רצים במצב קernel ומודדים להוות סכנה למחשב (יש הרבה מאוד דרייברים ואי אפשר לבדוק שאף אחד מהם הוא לא וירוס). לשם כך הומצאו פריבילגיות ביןיהם - בין המשמש במצב קernel - אף על פי שלא משתמשים בזיה ברוב מערכות הפעלה וכן מחשבים קצת יותר חשובים לנזקים.

התקני I/O

כדי לחבר התקן קלט/פלט כגון עכבר, מקלדת, דיסק משתמשים ב-System Bus שבאמצעות דרייברים לכל מכשיר מחבר כל התקן באופן מסודר ונגיש.

שימוש ב-System Bus מאפשר חיבור של התקנים חדשים בקלות הורדת Controller שלהם שמאפשר להתmeshק איתם וזהו. בנוסף, על אותם כבילים אפשר להתקין (באמצעות הגנרטו) הרבה התקנים שונים ולא צריך אחד מיוחד לכל אחד.

עם זאת, יש לו כמה חסרונות. בעיקר, הוא יכול להוות צוואר בקבוק כי המהירות שלו מוגבלת ע"י אורכו, מספר התקנים המוחברים אליו והדרישה שתואים להרבה התקנים שונים וכן הוא צריך לתמוך במהירות העברה שונות.

ה-Bus מכיל שלושה סוגים מידע שעובר בכבילים: חוטי כתובת (איפה נכתב את המידע), חוטי מידע (מה המידע) וחוטי אות (מתאים את קצב הטעינה).

כדי לתקשר עם התקני I/O משתמשים בקונטロלייט שלהם שנדרקים כשותבים למקומות ספציפיים שמוגדרים להם. הקונטロולר רץ בפרק עם DMA (הסכמה למעבר מידע מהמעבד ל זיכרונו מההרצתה הקודמת).

כשהתקן מסיים את העבודה על הנתונים שהוא קיבל הוא צריך לבקש את תשומת ליבו של המעבד (אם מקלדת אז כדי להעביר נתונים, אם מדפסת אז כדי לקבל משהו להדפס), וכך להתריע למעבד (דרך ה-OS) על זה, משתמשים ב-interrupt-ים.

אינטראפטים

gorom למעבד לעזור את מה שהוא עשו עכשו ולהריץ את פ' של ה-OS שמתמודדת עם ה-interrupt. התנהלות עם interrupt תפקידה של ה-OS.

נשאלת השאלה איך החומרה, ההתקן, יודע לאיזו פ' לקרו? מה אם היא לא הייתה קיימת כשיוצר המעבד? לשם כך יש וקטור **אינטראפט** - זה מקום קבוע בזיכרון שנקבע ע"י החומרה שבו ה-OS צריך לשים פיניטרים לפ' לכל אינטראפט בהתאם לאינדקס שלו (אינטראפט 39 יקרא לפ' במיקום 39 בוקטור הזה). כל פעם שמערכת הפעלה נדלקת והתקנים חדשים מותקנים ה-OS שמה את הפיניטרים הרלוונטיים בוקטור.

הערה הוקטור נחיש חלק בלתי נגייס בזיכרון למשתמשים כדי למנוע שימוש לhicנס למצב קרנל באמצעות אינטראפטים מכונים.

החוואר תיקח באופן עיוור את הפ' מהוקטור ובאותו הזמן תעבור למצב קרנל. חלק מהמנגנון, מזהים את ההתקן באמצעות מספרי זהות לדוגמה וכן בנסיבות ניתן להתmeshק עם התקן ב-OS.

בזמן שטפלים ב-interrupt מתעלמים מ-interruptים, אלא אם הם קריטיים - Non-Maskable (בלתי מסיכים) - ולא נרצה לשכוח מהם כגון כיבוי המחשב. בנוסף, אם ה-OS עושה משהו חשוב כרגע שהוא יכול להפסיק היא יכולה להגיד שהיא לא רוצה לקבל אינטראפטים. נזכר כי יש שלוש דרכים לhicns למצב קרנל: אינטראפט (אינטראפט חומרה); סיס科尔 (אינטראפט תוכנה); שגיאות (טכנית גם אינטראפט תוכנה). אף על פי שאנו מתייחסים אליהם כמשהו שונה, הם למעשה אותו הדבר וכולם **משתמשים באינטראפטים** (בפרט בוקטורן האינטראפט), כאשר trap הוא למעשה אינטראפט מספר 0x80 בקומונציה שגורא ל-syscall.

סיווג אינטראפטים

- תוכנה (טראף ושגיאות) מול חומרה (אינטראפטים של התקנים).
- פנימי (תוכנה) מול חיצוני (חוואר).
- סינכרוני (זמן תיק של השעון) מול אסינכרוני (חיצוני).
- מסיך (מלשון מסכח, אינטראפטים פחות חשובים) מול בלתי-מסיך (אינטראפטים קריטיים).
- מחזורי (השעון) מול לא מחזורי (כל השאר).

לא כל מה שה-OS חייב לרוץ במצב קרנל וככל שיש לנו קרנל יותר קטן כך יש פחות תקורה אבל פחות אבטחה כי לא הכל נעשה במצב בטיחותי שבו הכל נבדק. אם הקרנל קטן במיוחד אז יתפס פחות מקום בזיכרון ואם משנים פ' מחוץ לקרנל, לא צריך להתקן את כל הkernel מחדש אלא רק החלקים החיצוניים.

בעבר מעדים היו **מונייטיים**, כלומר הכל היה בקרנל. מעבדים יותר חדשים הם **מודולריים-מונייטיים**, כלומר קרנל יחסית קטן וכשצריך סקריפט נוסף הוא נטען מספירה חיצונית. יש בנוסף **מיקו-קרנל** שמקיל את המINIומות הנדרש ומשאיר את כל ניהול הקבצים וכו' לפ' שלא דורות שוכנת בזיכרון. כך הבטיחות יורדת בהרבה (ככל ארוחי פרחי יכול לכתוב לקובץ), אבל התקורה קטנה בהרבה. במקרים שהכל יהיה בפנים, יש "סרברים" שמשרתים את המINIות המקבילות למה שהיה קורה בתוך הkernel (סרבר ניהול קבצים, סרבר זמן תחזוקה תהליכיים). מיקרו-קרנל לא באמת שימושי במציאות והוא בעיקר מושך מחקר מעניין.

תרגול

יש פערים בסיבוכיות של כל מיני פעולות בין מבני נתונים עליים לבין איך ה-OS עובדת. בשני קטעי הקוד הבאים לכאהרנה נדמה ששנייהם בעלי סיבוכיות זהה, אך למעשה בגל האופן שבו המעבד עושה cache לבלוקים של קוד אחרי שהשתמש בהם לראשונה, הגישה הראשונה תהיה מהירה הרבה יותר כי בגישה השנייה הוא כל פעם ניגש למערך אחר (מערך זו ממש הוא מערך של מערכיים רציפים).

Approach 1	Approach2
<pre>for (h=0; h<height; ; ++h){ for (w=0; w<width; ++w){ img[h][w] = 0; } }</pre>	<pre>for (w=0; w<width; ++w){ for (h=0; h<height; ; ++h){ img[h][w] = 0; } }</pre>

איור 5 : הדגמה של סיבוכיות לא טריומיאלית

גם בהשוואה בין וקטור לרשותה מקוורת יש פערים גדולים מאוד בסיבוכיות פרקטית כי הוקטור יטמין את המערך מוקדם מאוד בעודו ברשותה צריך לעשות גישה לזיכרונו כל פעם מחדש.

בנוסף, הסיבוכיות של איפוס ה-RAM הואlianari באורך ה-RAM, שהוא אפקטיבית קבועה אך למעשה אין.

הגדירות

1. תכנית היא קובץ שנitin להרץ.
2. תהליך הוא הרצה של תכנית. תהליך פעיל הוא התהילה שהמעבד ברגע מרץ.
3. קלט/פלט הוא אוסף ההתקנים, תכניות ופעולות שאמונה על העברת מידע מ- או לצידם פריפרלי.
4. גרעין הוא יסוד מערכת הפעלה ויש לו שליטה מלאה על כל מה שקרה במחשב. בקורס השתמש ב"מערכת הפעלה" ו"גרעין" לשינויי אפ"פ שהם שונים במציאות (גרעינים לא מונוליטים). הגרעין הואאמין וכל תוכנה שאינה הגרעין היא לא אמינה.

מה שלוקח הרבה זמן זה לעבור לertz קרנל ממצב משתמש ועושים את זה כאמור עם אינטראפט.

לכאהרנה כדי לבדוק האם התקן רוצה להעביר/lקלבל מידע מהמחשב אפשר לעשות לולאה שבודקת את כל ההתקנים כל פעם אם הם רוצים משחו, אבל לשיטה זו שני חסרונות מרכזים : זה בזבזני מבחינת סיבוכיות וזה גורם לאובדן מידע במקרה שהשעונים של המעבד וההתקן לא מסוכנים. במקרה זאת, נmana את ההתקנים להודיעו כשהם מסיימים משימה.

האינטראפטים מנוהלים על ידי ה-PIC, Programmable Interrupt Controller - PIC, שמחובר מצד אחד להתקנים בחוטים ייעודיים שמעבירים לו את אותן ומצד שני למעבד בחוט שמעביר את המספר של ההתקן שהקפיץ את האינטראפט.

איןטראפט הוא למעשה APC, Asynchronous Procedure Call - APC, קריאה אסינכרונית לפרקודה - שכן האינטראפט בלתי נשלט ע"י התכנית ולכן אסינכרוני. האינטראפט בלתי נראה לתכנית שהופסקה כי היא פשוט מפסיקה (אול) חוזרת כמשמעותי לטפל באינטראפט.

שלבי האינטראפט

- 1. קבלת האינטראפט**: מגע אות חשמלי למעבד בסוף מחזור של המעבד ולכון לא נוצר פקודה של המעבד באמצעות (חיבור לדוגמה).
- 2. שימירת המצב הנוכחי**: נשמר את ה-PC וכל האוגרים האחרים של התכנית שריצה עד עכשו.
- 3. העברת שליטה וסיפוק הבקשה**: מעבד אוטומי למצב קרנל וקריאה לפ' הרלוונטי מוקטור האינטראפטים.
- 4. החזרת המצב הקודם**: נחזיר את PC ושאר האוגרים.
- 5. החזרת השליטה**: חוזרת למצב משתמש והחזיר התכנית המקורית.

דוגמה נבצע סכימה, חיסור ופעולות "וגם" ביטית ובמציע החישור נזוז את העכבר, אז המעבד יסיים את החישור קודם, יזוז את העכבר ויחזור ולעשות את פעולה ה-"וגם".

אחרי שmotkbl אינטראפט שגיאה, נותנים ל-**handler**, לנסות "לפטור" את הבעיה ולא במקרה של page fault הוא יחרוג את התהיליך.
דוגמה האינראפט צמום והוא פקודה שעוצרת את פעולה התכנית ונוננת את השליטה לדיבאג', זה למעשה **breakpoint**.

שבוע III | תהליכיים וחוטים

הרצאה

הגדולה תהליך הוא מופע של ריצת תוכנית.

תוכנית היא טקסט, ותהליך הוא אובייקט של OS שנעשה דברים. כשמייצים תוכנית יוצרים תהליך שחי לאורך מהלך הריצה.

איך תוכנית הופכת לתהליך

- כתיבת קוד המקור (פקודות ב-**c**).
- קימפול קוד המקור-**l-e** (פקודות לפי הארכיטקטורה של המעבד, לדוגמה 86x). ה-**l-e** מכיל גם מידע ש策יך לתחילת המעבד לפניה תחילת התוכנית.
- טעינת תוכן הקובץ אל תוך הזיכרון.

הערה נניח מעתה שיש לנו מספיק זיכרון לעשות מה שנרצה.

חלקי מרחב הכתובות של התהיליך

כל תהליך יש חלק בזיכרון שלו נראה רציף (אך אינו במצבות, נדון בכך בהמשך) שמורכב מארבעה חלקים.

- 1. טקסט - הפקודות של exe.a**.

2. מידע - נתונים סטטיים וערכיים מאותחלים מ-.exe.
3. ערמה - כאן מוקצת זיכרון באופן דינמי (גודל לא קבוע).
4. מחסנית - ניהול קריאות לפונקציות ומשתנים מקומיים (גודל לא קבוע).

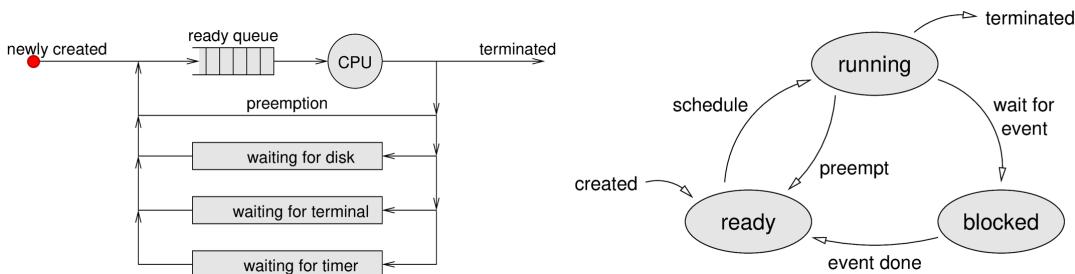
תכנית לא מדירה חח"ע תחיליך כי אפשר להריץ את אותה הוכנית כמה פעמים בו זמן, לדוגמה שרת שמקפל כמה דברים בו בזמן שירץ gcc אבל ברור שאלה תהליכיים שונים.

מצבי תהליכיים

אם נס镡ת ריצה באופן רציף, נזכיר כי מערכת הפעלה עוברת כל הזמן בין התהליכיים שרצים וכן תהליכיים לא תמיד רצים ויש להם מצבים אחרים.

1. רץ (running) - זה מה שהמעבד מבצעCurrently executing.
2. מוכן (ready) - יכול לרוץ אבל תחיליך אחר רץ כרגע על המעבד.
3. חסום (blocked) - לא יכול לרוץ כי הוא מוחכה למשהו (לדוגמה התקן קלט/פלט).

ברגע שתהיליך נוצר, הוא נכנס לתור המוכנים (ready queue) שמכיל את כל התהליכיים שמוכנים להרצה, מתי שהוא מתחילה באמת לרוץ על המעבד עד שהוא צריך למשהו (שעון, דיסק), מוחכה לדבר הזה ואז חוזר לתור המוכנים וחזור חלילה. אם הוא מסיים את הוכנית שהוא רץ, סורגים את התהיליך והוא יוצא מהולאה הזו (ראו הדוגמה).



איור 6 : הדוגמה של מחזור החיים של תחיליך

כשמדילקים את המחשב מתחילה תחיליך שמרץ shell או GUI (במקרה של וינדוס). כשכותבים פקודת או לוחצים על אפליקציה בהתאם, מתחילה תחיליך חדש בהתאם למה שנבחר.

הערה כל תחיליך יכול לקרוא ל-Syscall שיתחיל תחיליך חדש.

זמן שהטהיליך במצב רץ, מעודכנים באוגרים של המעבד את כתובות הזיכרון של התהיליך בזיכרון המשמש - בין היתר איפה מתחילה הזיכרון ומה גודלו, וכך יודע המעבד האם גישה לזכרון היא חוקית או לא (ריעונייה, במצבות זה קורה יותר מרכיב). בנוסף נשמר הפוינטර לראשית המחסנית של התהיליך הנוכחי וה-PC מופנה לפקודות של התהיליך הספציפי.

קונטקסט של תהליך

המכלול של המידע של התהליך נקרא הקשר של ההרצת (Execution Context) שנשמר בשלושה רבדים שונים.

1. **בתוכם המעבד :** האוגרים שהתהליך משתמש בהם, לרבות PC, SP, ו גם אוגר המידע והכתובת וכיו"ב.
2. **בזיכרו המשמש :** מרחיב הכתובות של התהליך שהוזכר לעלה.
3. **במערכת הפעלה :** מספר הזוחות של התהליך, המצביע עליו, המשתמש שMRI'ו אותו (כדי לדעת אילו הרשות יש לו), אילו קבצים פתוחים אצלו, מידע לניהול התהליך ועוד.

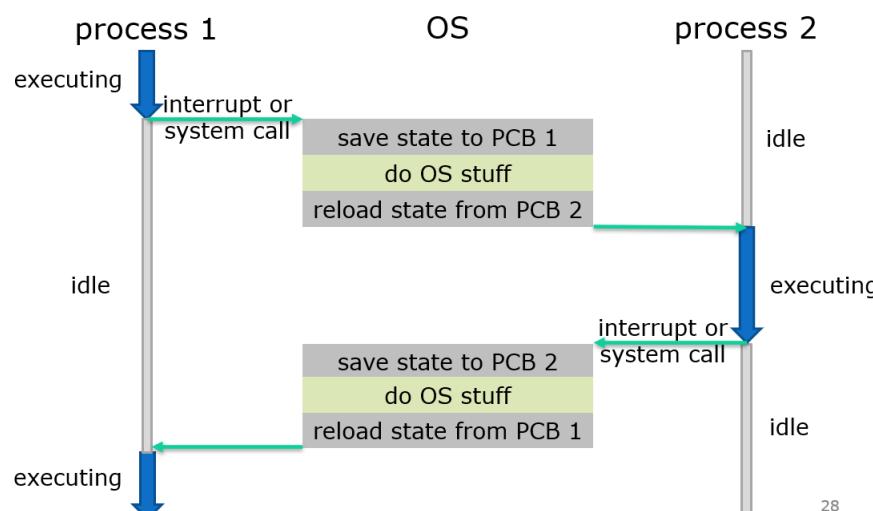
הקונטקסט הזה נשמר ב-OS במבנה נתונים מיוחד שנקרא (Process Control Block (PCB), והוא המוכנים ממומש כרשימה מקושרת בין PCB-ים).

ה-PCB שומר גם את מצב האוגרים במעבד, גם מיקום מרחיב הכתובות וגם את המידע של ה-OS על התהליך.
תהליך יכול להסתיים בכמה דרכים: יציאה רגילה או יציאת שגיאה (לדוגמא אם קיבל פרמטר לא טוב) - יציאות יזומות; שגיאה קטלנית (לדוגמא חלוקה באפס) או הריגת שגיאה מהתהליך אחר (בנסיבות אחרות) - יציאות לא יזומות.

Multiprocessing

לרוב במחשב ירצו כמה תהליכיים ביחד וזה OS צריך לדעת לנשל אותם, ובעיקר להמשיך את האשליה שככל תהליך רץ מטעם עצמו ולא מתחרה באף אחד (זיכרו נפרד לכל אחד, חולקת זמנים בין התהליכים).

כדי לעבור מהרצת תהליך אחד לאחר נצטרך לבצע נסיעה (Context Switching), כלומר לשמר את מצב המעבד הנוכחי, של כל האוגרים שלו ב-PCB, לעדכן את המידע ב-PCB של התהליך הקודם, להעביר את ה-PCB לתור הנכון (מוכן, ממחכה), לבחור תהליך חדש להרץ, לעדכן את ה-PCB של התהליך החדש ולהחזיר את המצב הקודם של המעבד והזיכרו מה-PCB.



איור 7 : הדוגמה של Context Switching

מה צריך לשמר כshawers בין תהליכיים? את הקונטקטס (איפה הוא מרחיב הכתובות שלו וכו') ואת האוגרים של המעבד אבל לא את הזיכרונו של המשתמש כי הוא לא משתמש.

מתי עושים קונטקטס סוויז'?

1. אינטראפטים : אם נגמר ה-timeslice של התהליך או שהתקן חיוני (שעון שביל להרץ סרטון בתמונות עם הפרשי זמן קבועים) הופך תהליך עם חשיבות גבוהה לモכו.
2. שגיאת זיכרון : כתובות הגעה לזכרון וירטואלי וסיבות נוספות שבחנו נדוע בהמשך.
3. מתוך התהליך : Syscall שדורש המתנה.
4. שגיאה : כשלורת שגיאה והתהליך נהרג.

מי שקובע איזה תהליך עושה את זה הוא ה-CPU Scheduler שעליו נלמד בהמשך והוא משנה קונטקטס מאות פעמים בשנייה. מי שמבצע את ההעתקה של המידע בין ה-PCB למעבד וכו' הוא ה-Dispatcher.

תקשורות בין תהליכיים

תהליכיים צריכים להיות נפרדים אחד מהשני כדי להגן עליהם, אבל לעיתים נרצה תקשורת בין תהליכיים. לשם כך ה-OS מספקת שירות שנקרא Inter-Process Communication (IPC) שבאמצעות תקורה גבוהה מאוד ודרך Syscalls מבצע את התקשרות זו. כדי למשמש את ה-IPC אפשר ליצור זיכרון משותף שיוכלו לגשת אליו כל התהליכים, או באמצעות "הודעות" ביןיהם (זהו שימוש ה-socketים שמכור מתקשורת אינטרנטית גם) או באמצעות קריאה וכתיבה מקובץ רגיל. מימושי IPC מחולקים לשתי קטגוריות - מבוססי זיכרון משותף ובסיסי הודעות. לא משנה איך ממשים את ה-IPC זה מוביל לביעות סכירותן. ככלומר האם התהליך קורא בזמנן את התוכן שMOVUBER לו או שהוא בדיק פיספס כי זה נדרס ע"י תוכן חדש וכו', וזה הנושא של השבוע הבא.

חוטים

חווט הוא מסלול ריצה בתוך התכנית. עד כה דיברנו על מצב בו לכל תהליך יש חוט שליטה יחיד שמתחליל ב-main ורך בהתאם לפקודות. עם זאת, ב-multithreading לכל תהליך יכולים להיות כמה מסלולי ריצה שונים חלקיים שונים של הקוד בו זמינים. בינו לבין תהליכיים, בין חוטים באותו התהליך אין הפרדות קונטקטס (למענה ההבדל היחיד הוא אילו פקודות רצوت).

דוגמה למה צריך חוטים שרצים בו זמינים? בדף לדוגמה, נרצה להתmesh עם המשתמש, לטענו תമונות ולהוריד דפים מהאינטרנטן כל זה בו זמינים. בדוגמא גם אפליקציה לכתיבת טקסט.

אם לא הייתה מקבילות, היינו נאלצים לעשות הכל אחד אחרי השני, כלומר ש-Word יזכה לתו ובנתים לא יעשה שום דבר, ואז יעבד קצת ויחזר לחכות לתוים וככלל לא מאדיע.

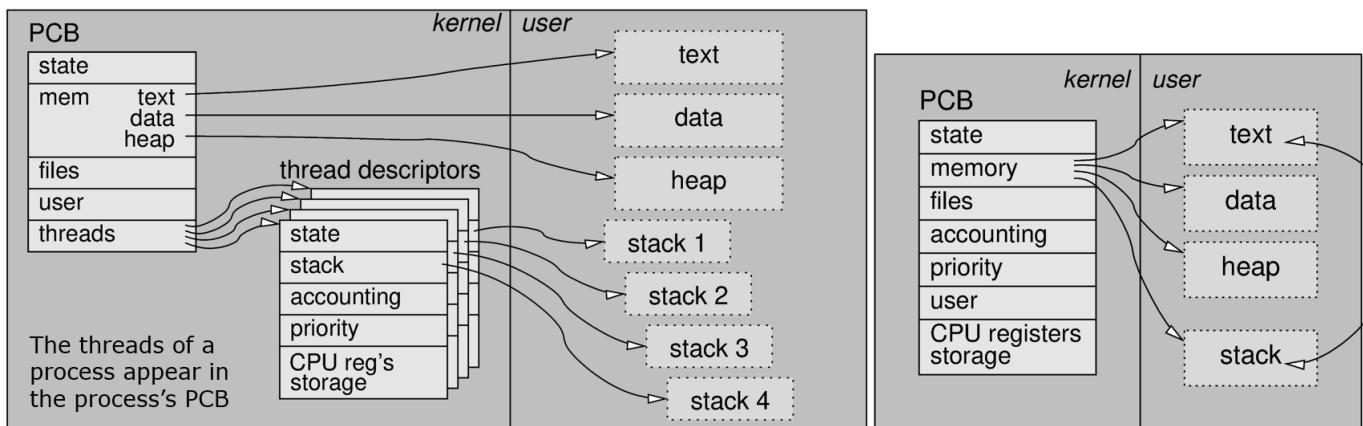
אם הייתה מקבילות ברמת התהליכים, ככלומר כל תהליך היה אמון על משימה אחת הייתה לנו תקורה מאוד גבוהה מבחינת ה-IPC שחייב לחבר בין כל המשימות ויש חזרתיות של הזיכרונו. הפתרון הוא להשתמש בחוט לכל משימה שחולק זיכרו עם חוטים אחרים.

תהליך הוא וירטואלייזציה של המחשב (לדעתו הוא היחיד שקיים) ואילו חוט הוא וירטואלייזציה של המעבד (במחשב עם כמה מעבדים יוכל להריץ כמה דברים בו זמנית, במקרה זה כמה חוטים באותו תחילה).

דוגמה במקרה של שרת, יהיה לנו חוט שמקשיב לפורט 80 לפי פרוטוקול http, שבכל פעם שיקלוט תקשורת מלוקה, יפתח חוט Worker שירץ פ' שתתפל בבקשת הלוקה (יתממשק עם קבצים בשרת וכו'). בזמן שהחוט העובד עושה את שלו החוט המקשיב יוכל לקבל בקשות חדשות ולפתח חוטים חדשים לכל לקוח חדש. כאן אחוריותה של מערכת הפעלה היא ליצור את החוטים, להחליפם בהרצתם, מנהל את התקשרות וכו'.

רמות חוטים

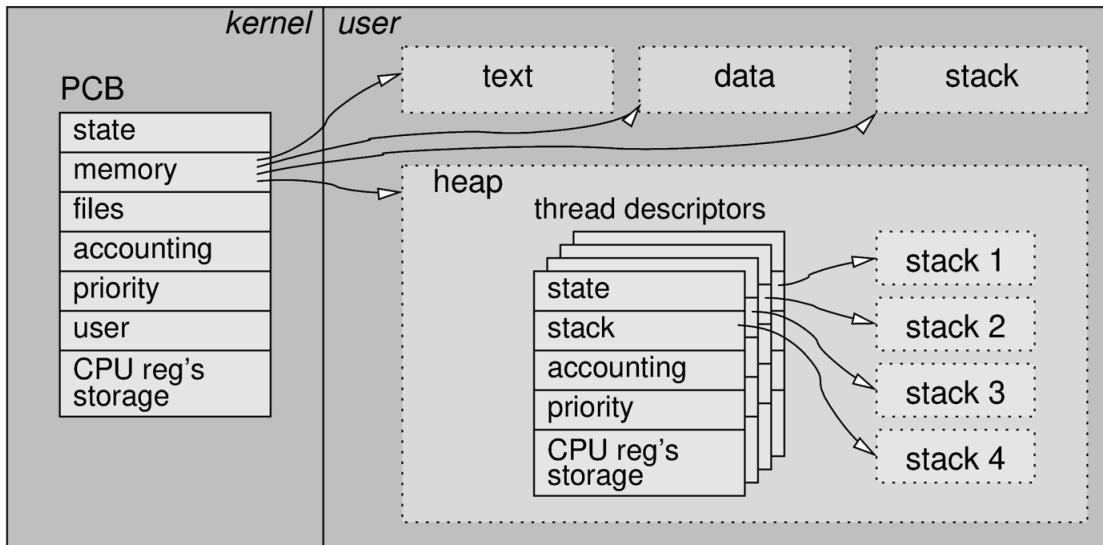
1. **חוטים ברמת הkernel :** במקרה כזה הקernel מנהל ושומר מידע על התהליך והחוטים ואפשר לחסום חוט אחד ולא לחסום חוט אחר (חוט אחד מוכחה לאינטרנט, אחר יכול לזרז בתנים). החסרונו של שיטה זו היא התקורה שנוצרת במעבר בין חוטים, שדורש את הקernel. אין להתבלבל בין **חוטי kernel**, חוטים שתפקידם ניהול מערכת הפעלה וחוטים **ברמת kernel**, שבסה"כ מנהלים ע"י הקernel אבל רצים במצב משתמש ותפקידם לנוהל את האפליקציה.



איור 8 : ניהול תהליכי עם תהליכי אחד (ימין) מרובת חוטים בשיטת רמת הקernel (שמאל)

2. **חוטים ברמת המשתמש :** במקומות שהחוטים יונחו ע"י הקernel, נמשח חוטים עם סדריות במצב משתמש והkernel יחשוב רק חוט בכל פעם והספריה תדמה קפיצה בין חוטים ותקטין את התקורה שקיימת במעבר בין חוטים ברמת הקernel. החסרונו של שיטה זו הוא שאם קוראים ל-Syscall `clone()` שמקפיא את החוט, הוא מקפיא גם את כל שאר החוטים כי לדעת הקernel יש רק חוט אחד וזה איפשר לעבור להריץ אחד אחר.

דוגמה פיתון עובד עם חוטים ברמת המשתמש, בשל מגנון **GIL** ש מגביל תהליכי שמרץ תכנית פיתון לחוט אחד בכל רגע נתון.



איור 9 : ניהול תהליכי במערכת מרובות חוטים בשיטת רמת המשתמש

הערה במערכות מרובות חוטים (כל מחשב כיוומ) על ה-CPU חוט אחד בכל פעם, במקום תהליך אחד. עם זאת, יכול להיות יותר ממעבד אחד במחשב (ואז אפשר להריץ כמה חוטים בו זמנית).
מערכת הפעלה בוחרת הן איזה תהליך להריץ והן איזה חוט להריץ בתחום התהליכי החוויא (בחירה בין חוטים ברמתו הUSR בלבד).

יתרונות של חוטים

1. הרבה יותר מהיר ליציר חוט חדש מאשר תהליכי חדש.
2. הרבה יותר מהיר לסיים חוט מאשר תהליכי.
3. הרבה יותר מהיר לעبور בין קונטיקסטים של חוטים בתחום תהליכי מאשר בין תהליכיים.
4. תקשורת בין חוטים היא פשוטה עם שיתוף הזיכרון ולא דורשת כניסה למצב קרבול ו שימוש ב-IPC.

<i>processes</i>	<i>Kernel-level threads</i>	<i>User-level threads</i>
protected from each other, require operating system to communicate	share address space, simple communication, useful for application structuring	
high overhead: all operations require a kernel trap, significant work	medium overhead: operations require a kernel trap, but little work	low overhead: everything is done at user level
independent: if one blocks, this does not affect the others		if a thread blocks the whole process is blocked
can run in parallel on different processors in a multiprocessor		all share the same processor so only one runs at a time
system specific API, programs are not portable		the same thread library may be available on several systems
one size fits all		application-specific thread management is possible

איור 10 : סיכום של הבדלים בין תהליכיים, חוטי קernel וחוטי משתמש

תרגול

סקרנו מחדש אינטראפטים, כשהדבר החדש היחיד הוא להסתכל על אינטראפט כהודעה ל-OS שהיא צריכה לתפוס שליטה ולפעול בתגובה לאיירוע כלשהו.

הגדירה אותן התפקידים בתהליך שקרה איירוע חשוב.

אותות גורמים לתהליכיים לעצור את מה שהם עושים (לאחר סיום מחזור המעבד) ומחיבבים אותם להתמודד עם האירוע מיד. כל תהליך יכול לקבוע איך הוא מתנהל בקבלה אותות.

אות נוצר ע"י מערכת הפעלה ונשלח לניהול אצל התהליך, בעוד אינטראפט נוצר ע"י חומרה/תוכנה ונשלח לניהול אצל ה-OS.

הערה: אותן ביוניקס מסוימים ע"י מספרים ויש להם שמות.

מה גורם לאות?

- קלט אסינכריוני מהמשתמש, לדוגמה Ctrl+C ב-shell שגורם לאות סיום ריצה של תהליך.
- תהליכיים אחרים וה-OS, לדוגמה אם נגמר טיימר שהתהליך התחיל.
- אינטראפט תוכנה, לדוגמה כזה שנגרם לאחר הרצת פקודה לא חוקית. במקרה כזה, ההרצאה תיצור אינטראפט תוכנה, הוא יגיע ל-OS, והוא תעביר אל התהליך אותן את המידע על ההרצאה הלא חוקית.

דוגמיה העברת אותן באמצעות המקלדת. Ctrl+Z משנה את התהליך Ctrl+C שומר את מצב התהליך בקובץ.

דוגמיה העברת אותן ב-comand line. פקודת kill עם מספר התהליך הורג את התהליך, פקודה fg עם מספר התהליך הורג תהליך מושחה.

דוגמה יש פ' ספריה, kill, שבאמצעותה אפשר לשולח אותן לתהליכי אחרים.

הערה יש אמות לשימוש כללי של אפליקציות, תחת השמות 2/SIGUSR1.

הערה strace עוקב גם אחרי אמות, לא רק פקודות.

דרכי טיפול באמות

1. התנהגות בירית מוחלט: התהליך יכול להסתיים, להתעלם, להשהות את עצמו או להמשיך הלאה.

2. התנהגות מותאמת: ניתן להגדיר התנהגות יעודית באמצעות פ' הספריה .sigaction.

ישנו כמה תהליכי שאי אפשר להתאים את הטיפול בהם, לדוגמה KILL ו-STOP.

הערה אם לא מתאימים handlers מיוחדים לאמות, המערכת תשמש ב-handlers בירית מוחלט.

Handler-ים מוגדרים מראש

.1 SIG_IGN - מתעלם מהאות.

.2 SIG_DFL - מריץ את הandler הדיפולטיבי.

הערה לכל אות יש הandler שמתאים לו.

אם אנחנו מבצעים משוח חשוב בתהליכי (מחיקת קבצים לדוגמה), נוכל לעשות(masking) לאותות כדי לסייע לנו לעשות את מה שאנו עושים נכון ואו לעסוק באות החדש. לחופין, אם אנחנו מטפלים באות חדש, נוכל להגיד לאחד מהם לחייב עד שנשים עם האخر. כדי להזכיר אילו אותן אנחנו חוסמים, משתמשים בפ' הספריה .sigprocmask

sigprocmask מקבל שלושה פרמטרים: מה הפעולה שאנו מבצעים כרגע (הוסף אותות לרשימה החסומים, הסרה); רישימת האותות שאנו מוסיפים/מסירים; ופונינט לשימירת רישימת האותות החסומים הישנה.

sigaction מקבל שלושה פרמטרים: מספר האות; מה ה-*action* החדש שמוגדר לאות (handler), אותן מתעלם ממנו ודגלים אחרים); ופונינט לשימירת ה-*action* הקודם.

מסכת האותות חלה רק במהלך שה-*handler* הנוכחי מותקן.

דוגמה נרצה כשהוא מקבלים SIGINT, Ctrl+C, להציג הודעה למסך.

```

#include<stdio.h>
#include <unistd.h>
#include <signal.h>

void catch_int(int sig_num) {
    printf("Don't do that\n");
    fflush(stdout);
}

int main(int argc, char* argv[]) {
    // Install catch_int as the
    // signal handler for SIGINT.
    struct sigaction sa;
    sa.sa_handler = &catch_int;
    sigaction(SIGINT, &sa, NULL);

    for ( ;; )
        //wait until receives a signal
        pause();
}

```

<14|1>orenstal@puma:~/Desktop/OS/ex2/demo% demo
 ^C Don't do that!
 ^Z
 Suspended
 <15|1>orenstal@puma:~/Desktop/OS/ex2/demo% demo
 ^C
 <16|1>orenstal@puma:~/Desktop/OS/ex2/demo%

איור 11 : קטע קוד שמנדרת handler לאות SIGINT (מיימי) וריצה שלו (מיימי)

הלואה בסוף ממחה כל הזמן לאותות לפעול עליהם.

סקרנו מחדש תהליכי PCB והחלפה ביניהם.

אף על פי שלוחטים יש זיכרון משותף והרבה משבבים אחרים משתפים עם חוטים אחרים בתהליך, יש להם עדין מחסנית ואוגרים משל עצמם, לרבות PC, שכן אלו נדרשים לרכיב עצמאית של קוד.

חותמים ברמת המשתמש

ממומשים ע"י ספריה שאחראית על התחלתם, סיומים וכיוצא"ב. יתרון שלא הזכרנו של חוטים ברמת המשתמש הוא שאופן התזמון של החוטים הוא תלוי אימפלמנטציה ואפשר להתאים אותו לצרכינו. העובדה שרק חוט אחד רץ בכל רגע נתון ולכן רק חוט אחד יכול לגשת למשאב משותף. מיניתרת את חלק ממנגוני הסינכרון שצריך כשייש יותר מחרוט אחד (שבשבועות הקרים נעסק בכך).

בספריה שומרים לכל חוט, שמתארת את החוט, המידע שצריך על הקונטיקט שלו וכו'. בימוש ברמת המשתמש יש בתוך החוט (הקרנלי) תור מוכנים של חוטים.

כדי למש ספריה כזו צריך למשוחה בין חוטים, ככלומר, עצירת פעלת החוט הנוכחי, שמירת מצבו וקפיצה לאחר. את זה עושים עם שומר את המצב הנוכחי) וכן `sigsetjmp` (משוחרר מצב מזיכרו).

sigsetjmp מקבל שני פרמטרים: המיקום אליו שומרים את קונטיקט המחסנית (מיקומה, גודלה) ומצב המעבד; ופרמטר שקובע האם שומרים את מסכת העבודות או לא. לאחר מכן נשחזר את המידע האגורר הזה באמצעות `siglongjmp`.

ערך ההחזרה שלו 0 אם הוא חוזר מיד או ערך שמוגדר ע"י המשתמש אם עכשו הגענו מ-`siglongjmp` (כלומר שיחזרנו הכל והמשכנו להריץ מאיפה שערכנו).

הפ' שומרת את את PC,SP, את מסכת האותות בהתאם לפרמטר כאמור ואות מצב המעבד, אך לא שומרת משתנים גלובליים (הם לא משתנים), משתנים מוקצים דינמית (הם גם לא הולכים לשום מקום כי הם בערימה שהיא משותפת) וגם לא ערכיהם של ערכים מקומיים כי אלו שמוררים במחסנית.

siglongjmp מקבל שני פרמטרים : המיקום בו שמוררים הקונטקסט של החוטו ; והערך שמוחזר ל-**jmp** כאמור, שהוא יחזיר עצמו גם. דוגמה בדוגמה הבאה מופיעים שני קטעי קוד זהים, כאשר אנחנו מתחילהים את הריצה מהחוטו הראשון. תם הוא מערך שגדיר את הקונטקסטים של החוטים בהתאם.

```

Thread 0:
void switchThreads()
{
    static int curThread = 0;
    int ret_val =
        sigsetjmp(env[curThread],1);
    if (ret_val == 5) {
        return;
    }
    curThread =
        1 - curThread;
    siglongjmp(env[curThread],5);
}

Thread 1:
void switchThreads()
{
    static int curThread = 0;
    int ret_val =
        sigsetjmp(env[curThread],1);
    if (ret_val == 5) {
        return;
    }
    curThread =
        1 - curThread;
    siglongjmp(env[curThread],5);
}

```

איור 12 : הדגמה של שימוש ב-**siglongjmp**-ו **sigsetjmp** בפ'

מה שקרה בReLU זהה : מתחילהים מחוט 0, מגיעים ל-**sigsetjmp**, שומרים את המצב הנוכחי עם מסכת האותות כי יש ערך 1, לא נכנסים ל-if כי ערך ההחזרה הוא 0, ואז עוברים קונטקסט לחוט 1, בהסתמך על המידע שנשמר מלפני על חוט 1. לאחר מכן, שומרים את המצב הנוכחי של חוט 1 עם מסכת אותן, לא נכנסים ל-if, ואז עוברים לקונטקסט של חוט 0, שנשמר פעמי אחר מכן. לאחר מכן, שומרים את המצב הנוכחי של חוט 1 עם מסכת אותן, לא נכנסים ל-if, ואז עוברים לקונטקסט של חוט 0, שנשמר פעמי אחריו מ-**jmp** **sigsetjmp**, כי עכשו יצאנו מ-**jmp** **sigsetjmp** כזכור. הפעם כן נכנס ל-if ונסיים את הריצה. החצים מסבירים מה קורה أولית יותר טוב מטיקסט.

דוגמה נעשה שאלת מבחן דומה לדוגמה הקודמת.

7) כמה פעמים הקוד הבא ידפיס hello?

```
#include <setjmp.h>
#include <stdio.h>

int main(int argc, char *argv[])
{
    sigjmp_buf jbuf;
    int i = 10;
    int ret_val = sigsetjmp(jbuf,1);
    if (ret_val == 0) {
        return 0;
    }
    i--;
    printf("hello\n");
    siglongjmp(jbuf,i);
    return 0;
}
```

- א) 9 פעמים
- ב) 10 פעמים
- ג) 0 פעמים
- ד) פעם אחת

איור 13 : קוד המקור של השאלה מהמבחן

התשובה היא ג', כי ערך החזרה מה-sigsetjmp הוא 0 כייצנו ישירות ולא חזרנו דרך siglongjmp ולכן נמצא מיד מהריצה לפניו כל הדפסה שהיא.

לקרנל יש טבלת חוטים (TCB) שבה הוא עוקב אחרי כל החוטים (ברמת הקרנל), גם כדי להריץ את אלג' התזמון. בעיקרונו כל עוד תחילה כלשהו קיבל זמן מעבד, הקרנל יחליף רק בין חוטים ברמת הקרנל ששייכים לתחילה הזאת, ככלומר למשה חוטים מתחilibים שונים אינט מוצויים בתחרות אחד עם השני אף פעם.

מתי נשתמש ברמת הקרנל ומתי ברמת המשתמש? במקרה אם התכנית היא CPU-bound, ככלומר עשויה הרבה פעולות אבל פועלות שונות, עדיף להשתמש בחוטים ברמת המשתמש כי אין בעיה של חסימה מהתקנים חיוניים, לעומת זאת אם התכנית היא I/O-bound, ככלומר תלואה הרבה בתקנים חיוניים, עדיף להשתמש בחוטים ברמת הקרנל.

בתרגיל נமמש מחלוקת ניהול חוטים ברמת המשתמש, כאשר לכל חוט יהיה שלושה מצבים אפשריים: רץ, מוכן וחסום, כאשר התזמון שלו יהיה הבסיסי ביותר, ככלומר שנענף קלlesi כך שхот ירוזך רק לאחר שככל החוטים שהיו מוכנים לפני רציהם. נעצור את הריצה של החוט לאחר תקופת זמן קבועה (קוואנטום) אם הוא לא ייחסם את עצמו קודם.

שבוע VII | סינכרון

הרצאה

בתקשורת בין תחilibים שונים, אם בוחרים להשתמש במשאב כלשהו משותף, נוצרים בעיות של סינכרון.

דוגמאות לבעיות סינכרון

דוגמה ספולר של מדפסת. המעבד בתוך המדפסת שומר מערך עם ג'ובים שהוא צריך לעשות, וכל פעם שפרוסס רוצה להדפיס משהו כל מה שהוא צריך לעשות זה job = job[IN]++) Spooler[IN] (מצבע לראש התור) כלומר אנחנו מודמים כאן תור עבודה. אם תהליך 1 מבצע job1[IN] Spooler[IN]++) job2[IN]. לאחר מכן עושים קונטקסט סוויז', תהליך 2 עושה את אותו הדבר רק עם job2[IN] אז הכל טוב ויפה.

לחולופין, ניתן כי תהליך 1 יבצע job1[IN] Spooler[IN]++) או יקרה קונטקסט סוויז' (שהוא לא שלו מתי הוא יקרה), תהליך 2 יעשה בעצמו job2[IN] Spooler[IN]++) מעולם לא יודפס. לאחר מכן עושים שוב קונטקסט סוויז', תהליך 1 מעלה את IN, שוב קונטקסט סוויז' ופרוסס 2 מעלה את IN ומסיים.

קרווآن שני דברים רעים : job1[IN] נמחק ונוצר לנו רוח מיותר. הספולר ייכה שהערך האחרון *לעתם* באפר שלו יփוך ל-*לא* *לעתם* ואז י Mishik לרווח, הבעיה עכשו היא שהרווח הזה, שהוא *לעתם*, מעולם לא ישנה כי אנחנו שמים עבודות רק בהמשך. כלומר הרשנו לגמרי את הספולר.

הבעיה הזו לא ייחדית לתהליכיים, היא קורת גם בין חוטים.

דוגמה נניח שיש לנו כמה חוטים עם משתנה גלובלי IN. בכתיבת *IN*-*C*, נקבל באSEMBLY שלוש פקודות שונות : קריאה מהזיכרון, אינקרמנט ברגייסטר וכתיבה לזכרון (ראו קוד).

```
|w r1, 100
•
inc r1
sw 100, r1
```

איור 14 : קוד אSEMBLY של אינקרמנציה ערך בכתובת 100

העובדת שזה מופרד גורמת לכך שנitinן לבצע קונטקסט סוויז' בין הפעולות. יכול להיות שהכל יעבד כרצוי (קרי הקונטקסט סוויז') יקרה לאחר סדרת הפעולות אבל יכול להיות גם לא, דוגמה שיקרא הדבר הבא :

- חוט 1 : קורא מהזיכרון.

- חוט 1 : אינקרמנט.

קונטקסט סוויז' (הרגייסטרים משתנים).

- חוט 2 : קורא מהזיכרון.

- חוט 2 : אינקרמנט.

- חוט 2 : כתיבה לזכרון.

קונטקסט סוויז'

- חוט 1 : כותב לזכרון.

וכך הענו בסופו של דבר למצב ש-IN עליה רק באחד, לא ב-2 כפי שרצינו - שזה חמור!

הערה באגים בסגנון זהה לא תמיד קוראים ולכן קשה לעלות עליהם באמצעות הרצת טסטים (יכול להיות ש-1000 פעמים הכל היה בסדר אבל אז ב-1001 הכל קורס אל תוך עצמו).

הפתרון לבעה כזו הוא סינכרוניזציה, ונובע ממכב מציאותי (הניבו כן כי כל שורה באה אחרי קודמותיה).

דוגמה

אני	השותפה שלי
חוור הביתה	
מסתכל במקrror ורואה שאין חלב	חולך למכولات
חוורת הביתה	
מסתכלת במקrror ורואה שאין חלב	מגיע למכولات
הולכת למכولات	קונה חלב
מגיעה הביתה ושם את החלב במקrror	
קונה חלב	
מגיעה הביתה ומגלחת יש יותר מדי חלב	

טבלה 3 : הדוגמה של בעיית סינכרון במציאות

הפתרון כאן הוא לדוגמה לשים פתק במקrror ולהתריע שהלכתו לקנות חלב כך שהשותפה לא תצטרך לעשות כן. חשוב בכך להסביר את הפתק בסוף כדי שלא יחשבו שמשיחו קונה חלב כשנגמר כזה לא המצב. העיקרון הזה ינחה אותנו גם בסינכרון תהליכיים.

הבעיה היא שיש גישה לא מתואמת למשאב מסוות (זיכרנו, התקן חיצוני וכו'), לא משנה אם זה חוטים, תהליכיים או אם הם רצים במקביל או לא.

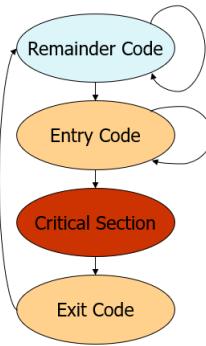
החלק הקרייטי בהתנגשות הוא הגישה לזכרו המשותף (כל שאר הפעולות הלוואקליות לא מעניינות). קטע הקוד הקרייטי יכול להיות קצר מאוד (אינקרמנט) או מאד ארוך. כדי שהכל יעבוד תקין, נדרש לכל היותר חוט אחד מ裏ץ קטע קוד קרייטי.

הפתרון הוא אלג' מניעה הדדיית (Mutual Exclusion) שמנע ריצה במקביל של קטעי קוד קרייטיים.

הערה יש בעיות סינכרון שאין קשרות למניעה הדדיית, לדוגמה קוד שדורש שקוד אחר יסיים לפניו - זו בעיית קורדינציה.

המודל של Dijkstra

כדי לפטור בעיה זו פורמלית, דיקסטרה יצר מודל מופשט של קטע קוד שמשמעותו בגישה למשאב מסוות. באior, רואים את קוד ה-*Reminder*, שהוא לא מעניין וועסוק בקוד לוקאלי שלא רלוונטי לבעה. לפני ואחרי החלק הקרייטי, יש שני קטעי קוד - **הכניסה**, שיכול לחתך הרבה זמן (לכן יש לו לפחות סיבוב עצמה) לדוגמה כשהוא מוחכה לחוטים אחרים, או להיות מאוד מהיר וקטע היציאה שרץ וזה חוות לקוד "לא מעניין". האופן שבו החצים מכונים מאפשר לקטע הקוד הקרייטי לרוץ פעמיים, הרבה פעמיים ושלפניהם ואחריהם יקרו כל מיני דברים - כמו שיותר גנרי.



איור 15 : המודל של Djikstra לבעיית מניעה הדדית

דרישות לאלג' מנעה הדדית

המשימה שלנו היא לכתוב את קטעי הקוד של הכניסה והיציאה כך שהכל יעבוד כמו שצריך. לשם כך יש כמה תכונות נדרשות:

1. מנעה הדדית: בכל רגע נתון רק תהליך אחד מרץ את החלק הקרייטי שלו (חוקיות הקוד).
2. התקדמות: כל תהליך שמנסה להגעה לקטע קוד קרייטי בשלב כלשהו הגיע אליו הוא או תהליך אחר (לא נוכל למנוע מכולם לגשת למשאב פשוט).
3. היעדר מרעב: אם תהליך מנסה להגעה לחלק קרייטי, הוא מתишחו הגיע אליו (לא יעוף אותו).
4. כלליות: אין הנחות על מהירות או מספר התהליכיים שרצים להריץ את קטע הקוד הקרייטי.
5. איסור חסימה ב-remainder: אף תהליך לא יכול לחסום אף אחד אחר מחוץ לקטע הקוד הקרייטי.

הערה אם קטע קוד הקרייטי זורק exception או קטע הייחודה טיפול בו ולא יחסום אחרים.

הערה יש כאן הנחה סטומה שתהליך לא נגמר באמצע קטע הקוד הקרייטי ורק לפרק זמן סופי.

פתרונות למנעה הדדית

1. קטע הכניסה חוסם את כל האינטראפטים וקטע היציאה מוחזר אותם. תנאי 1 מתקיים כי אף אחד לא יכול להתעורר, 2 מותקים בבירור, 3 מותקים כי מרים אותם לפי הסדר ו-5 מותקים כי הרימיניינדר בסדר גמור.
- עם זאת, 4 לא מתקיים כי הוא לא מתייחס למצב שבו יש לנו כמה ליבוט. עם זאת, באופן יותר פרקטני, מי שמשמש את האלג' הזה הוא המשתמש וממשתמש לא יכול לחסום אינטראפטים. גם אם מערכת הפעלה עשויה את זה היא צריכה לעשות את זה לפרק זמן מאוד קצר (לדוגמה בעת טיפול באינטראפט אחר כזוכו) ולא לפרק זמן ארוכים כלשה. מיסוך אינטראפטים כזה מקופה את המערכת למשך לפרק זמן לא ידוע.

2. נגידר משתנה גלובלי `turn` בגודל בית אחד שאי אפשר לגעת בו אלא בקטעי קוד הכניסה והיציאה
באופן הבא :

<table border="1" style="width: 100%; border-collapse: collapse;"> <tr><td style="background-color: #d9e1f2;">Reminder</td></tr> <tr><td style="background-color: #ffd700;">while(turn==1);</td></tr> <tr><td style="background-color: #c8512e;">Critical</td></tr> <tr><td style="background-color: #ffd700;">turn=1;</td></tr> </table>	Reminder	while(turn==1);	Critical	turn=1;	<table border="1" style="width: 100%; border-collapse: collapse;"> <tr><td style="background-color: #d9e1f2;">Reminder</td></tr> <tr><td style="background-color: #ffd700;">while(turn==0);</td></tr> <tr><td style="background-color: #c8512e;">Critical</td></tr> <tr><td style="background-color: #ffd700;">turn=0;</td></tr> </table>	Reminder	while(turn==0);	Critical	turn=0;
Reminder									
while(turn==1);									
Critical									
turn=1;									
Reminder									
while(turn==0);									
Critical									
turn=0;									
Code for Thread 0	Code for Thread 1								

איור 16 : קטעי הכניסה והיציאה עם משתנה גלובלי `turn`

כלומר, חוט 1 מוחכה עד ש-`turn` הוא 0, ככלומר הגיע התור שלו, מרים את הקטע הクリיטי ו מעביר את התור לחוט 1. חוט 1 עושה פעולה דומה וככה הם מעבירים ביניהם את התור.

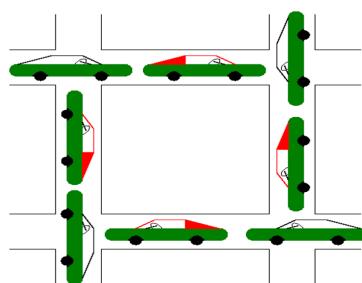
מניעה הדדית מתקיימת כי נחכה עד שייהי תורנו. איסור החסימה ברימינדר לא מתקיים כי אם חוט 1 קיבל את תורו אבל אף פעם לא נכנס לקטע קוד קרייטי אז חוט 0 נחසם לתמיד. לכן גם התקדמותו וגם היעדר רעב לא מתקיימים.

3. נגידר שני משתנים חדשים : `flag[0]` ו-`flag[1]` שהם הדגלים של חוט 0 ו-1 בהתאמה. אם אנחנו רוצחים להריץ את קטע הקוד הクリיטי, עללה את הדגל שלנו ונחכה שהחוט האחר יסייע. כך נתריע שאנחנו רוצחים לגשת ולא נצטרך להכחות שמותישחו יזכרו לנו (ראו איור).

<table border="1" style="width: 100%; border-collapse: collapse;"> <tr><td style="background-color: #d9e1f2;">Reminder</td></tr> <tr><td style="background-color: #ffd700;">flag[0] = true; while(flag[1]);</td></tr> <tr><td style="background-color: #c8512e;">Critical</td></tr> <tr><td style="background-color: #ffd700;">flag[0]=false;</td></tr> </table>	Reminder	flag[0] = true; while(flag[1]);	Critical	flag[0]=false;	<table border="1" style="width: 100%; border-collapse: collapse;"> <tr><td style="background-color: #d9e1f2;">Reminder</td></tr> <tr><td style="background-color: #ffd700;">flag[1]=true; while(flag[0]);</td></tr> <tr><td style="background-color: #c8512e;">Critical</td></tr> <tr><td style="background-color: #ffd700;">flag[1]=false</td></tr> </table>	Reminder	flag[1]=true; while(flag[0]);	Critical	flag[1]=false
Reminder									
flag[0] = true; while(flag[1]);									
Critical									
flag[0]=false;									
Reminder									
flag[1]=true; while(flag[0]);									
Critical									
flag[1]=false									
Code for Thread 0	Code for Thread 1								

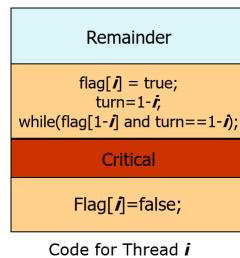
איור 17 : קטעי הכניסה והיציאה עם דגלים

מנעה הדדית מתקיימת, אין חסימה ברימינדר אבל אין התקדמותו ולכן גם לא היעדר הרעבה. לדוגמה, אם 0 מרים את הדגל, עשויהكونטקסט סוויז' או 1 מרים את הדגל אז אף אחד מהם אף פעם לא יגיע לקטע הクリיטי. מצב כזה נקרא דד-LOCK, כי שני החוטים מחכים אחד לשני. זה דומה לצומת לא פנוי (ראו איור). הפתרון בפתרונות הוא רמזוריים.



איור 18 : הדוגמה של דד-LOCK

4. נסעה לטפל בבעיה זו בדרך הבא: נוסיף חוזרת את מזען באופן הבא (נ' מقلיל לנו את שני המקרים): בכל פעם שSEGMENTS לקטע הכנסה, נרים את הדגל שאנו רוצים להציג חלקו הクリיטי, ניתן לאחר התוור וначלה שיגיע תורנו גם שהחוט האחר לא ררים את הדגל שלו, ואז ניכנס. בסוף נוריד את הדגל שלנו. למעשה אנחנו קודם פותחים את הדלת לאחר, מציעים לו להיכנס, בהתאם להחלטה שלו בסופו של דבר ניכנס גם אנחנו (אם הוא רוצה להיכנס נחכה לו ואם הוא לא רוצה נרוץ בעצמו). לא נגיעה לחילופים אינטנסיביים (הסטודנטית המשקיפה תריץ מקרה בו יש קונטיקסט סוויז' בamuance ומגלה שהוא אכן המנצח).



איור 19 : האלג' של Peterson

הפתרון הזה הוא מאד אלגנטיו ולמעשה מספק את כל חמישת התנאים.

הבעיה איתתו היא שהלולה בא אנחנו מוחכים לתהיליך אחר היא מיותרת וمبזבזת את משאבי המעבד כי המזומן לא יודע שהתחילה מבזבזו כוח חישוב. הבעיה השניה היא שאי אפשר להרחב אותו ליותר שני חוטים.

كونספטיים בתזמון

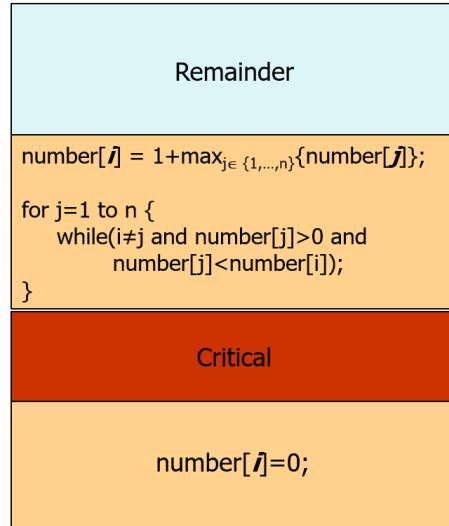
.1. מצב שבו בהינתן סדר אחר של תזמון הינו מקבלים תוצאה אחרת, קשה מאוד לעלות על מצבים כאלה. Race Condition.

.2. Atomic Instruction : פעולה שבוצעים בלי יכולת לקרוות קונטיקסט סוויז' במהלך. אינקרמנט של משתנה זה לא כמו שראינו אבל כתיבה לזכרו כן.

.3. Busy Waiting : בדיקה חוזרת של תנאי ובזבוז משאבים באותו הזמן. הפתרון לדבר זה הוא לחכות לאינטראפט שמספר לי שהתנאי חדל מלתקיים ועד אז לא לרוץ בכלל.

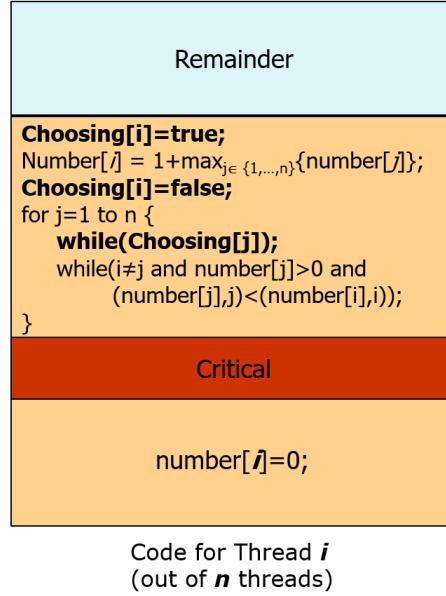
ניסיונות לאלגוריתם המאפייה של למפורט

נניח שאנו במאפייה שבה כל אחד לוקח מספר ומחכה בתור (לא בישראל). קטע הכנסה הוא לחת מספר ולהחות עד שיגיע תורנו וקטע הייצאה הוא לעדכן את המספר הבא בתור על המסך. האלג' עובד באופן אינטואיטיבי יחסית, רק שהבעיה כאן היא שלKİחת מספר אינה אוטומטית ואז אין מניעה הדדית. נסעה למשמש אותו עם שינויים כדי שייעבוד.



איור 20 : מימוש ראשון של אלג' המאפייה

1. בוחרים את הערך הראשון אחרי כולם ואז בודקים לכל חוט אחר שונה מアイתו שמחכה בתור שהוא לא לפניו, ורק כשכלם לא לפניו נכנסים. אנחנו כאן סומכים על זה שמי שיקח מספר אחרינו (כלומר הוא סיים את תורו ואולי חוזר חזרה לתור) הוא בטוח אחרינו ולא צריך לבדוק אותו שוב.
2. אם היוו משנים את האי-שוויון להיות חלש, כלומר נחכה שכל אלה עם מספרים כמוינו ילכו לפניו נקבל בבירור דד-лок.
3. כדי לפטור זאת, נבדוק עבור שניים באותו המספר למי יש מספר חוט יותר גבוה. הבעייה כאן היא שלקחת מספר דורשת קריית ערכים, חישוב מקסימום וכטיבת ערכים. יכול לקרותו כאשרRace Condition כי חוט 4 יכול לחשב מקסימום אבל עוד לא כתוב לזכורו, יקרה קונטקט סוויצ' וחות 5 יחשב את המקסימום, יסיים גם לכתוב ויכנס לקטע הקוד הクリטי. אז יקרה עוד קונטקט סוויצ', חוט 4 יראה שיש לו מספר זהה לאחר אבל בגלל שהוא לפניו במספר החוט הוא יכנס גם ויש לנו עכשו שניים בקטע הקוד הクリטי בניגוד לתנאי המניעה ההגדית.
4. הפתרון הסופי לכך הוא להרים דגלי בחירה, כלומר כמשמעותם מקסימום להרים לפני ולהורד אחורי את הדגל "אני בוחר" וכשבודקים בלולאת-while האם יש מישחו לפני, נחכה שכל חוט אחד יסיים לבצע מספר (אם זה מה שהוא עשה קודם, ראו אייר).



איור 21 : מימוש נכון של אלג' המאפייה

האלג' כאן גם שומר על FIFO (למעט עבר שניים שבוחרים את אותו המספר).

פעולות אוטומיות בארכיטקטורה מודרנית

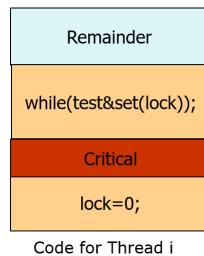
קיימים במעבד יש פעולות אוטומיות מורכבות מאוד כדי לאפשר פעולות בסיסיות בלי קונטיקסט סוווי' במאצע.

• שימוש שמקבלת בית lock, הופכת אותו ל-1 אם הוא 0 ומחזירה את ערכו הקודם בפעולה אוטומית אחת.

• שימוש שמקבלת פוינטר ומספר ומוסיף אותו למספר בפויינטר ומחזירה את ערכו הקודם של המיקום בזיכרון (בפעולה אחת).

• שימוש שמקבלת פוינטר, ערך ישן וחידש שמה בפויינטר את הערך החדש רק אם לפני היה את הערך הישן ומחזירה האם הצלחה או לא.

אפשר למשבב באלג' המאפייה במאצעת Test&Set באמצעות Test&Set (ראו איור).



איור 22 : מימוש עם Test&Set של אלג' המאפייה

מה שקרה כאן זה שאנו מוחכים עד שהLOCK פתוח לשימושינו, מבצעים את קטע הקרייטי ובסיומו משחררים את הLOCK. הבעיה כאן היא שאין היעדר הרעבה כי האם תפנסנו את הLOCK כשהוא פנוי הוא על בסיס מזל כי יכול להיות חוט אחד שיחכה אינסוז' זמן כי הוא אף פעם

לא הגיע בזמן לлок פתוח. ניתן לפתור זאת באמצעות האלגי' של Burn, שבודק כמה זמן כל אחד חיכה ועוד דברים מורכבים אחרים שלא עוסק בהם.

פתרנו את בעיית המניעה החדדית אבל עדיין לא פתרנו את ביזבוז משאבי המעבד. כדי לפתור את זה נבקש את עזרת ה-OS באמצעות פרימיטיבים סנכרים עם משקל ברור. הפרימיטיבים האלה ממומשים בספריות (השלב הכי גבוה שהוא לא אפליקציוני). עוסק בפרימיטיבים באופן אבסטרקטי בלי להתעניין איך הם ממומשים מבחינה פנימית.

תרגול

נרצה להשתמש בהרבה חוטים או תהליכים אם יש לנו הרבה מעבדים, אם יש לנו פעולות חסומות או אם יש לנו כמה פעולות שצורך לעשות בו זמן קצר. להלן טבלת תוצאות של ההבדלים בין תהליכים, חוטי קernel וחוטי משתמשים.

	Processes	Kernel Threads	User Threads
Interaction between instances	protected from each other, require operating system to communicate	share address space, simple communication, useful for application structuring	
Context-switch overhead	high overhead: all operations require a kernel trap, significant work	medium overhead: operations require a kernel trap, but little work	low overhead: everything is done at user level
Blocking granularity	independent: if one blocks, this does not affect the others		if a thread blocks the whole process is blocked
Multi-core utilization	can run on different processors in a multiprocessor system		all share the same processor
OS dependency	system specific API, programs are not portable		the same thread library may be available on several systems
Scheduling	one size fits all		application-specific thread management is possible

איור 23 : הבדלים בין תהליכים לבין חוטים

ספרייה pthread

כל פעם שתתחליך נוצר ייש לו חוט ראשוני, וכל השאר צריך LICOR בפונקציית pthread_create. כדי לעשות זאת, משתמשים בספריית pthread.

pthread_create מקבל פוינטר לשמיירת מספר החוט; אטריבוט לא מעניין; מה להריץ ברגע שהחוט נפתח; ואילו ארגומנטים להעביר לפ' הזו. היא מחייבת 0 אם הצליחה ואחרת את קוד השגיאה.

כדי לסייעים חוט יש כמה אפשרויות.

- קוראים ל-pthread_exit עם פרמטר סטטוס סיום שהיה נגיש לחוט שירץ עלייו pthread_join.
- אם החוט מסיים את ריצה הפ' שהוא קיבל בהתחלה.

- אם הוא מבוטל עם pthread_cancel שמקבל את מס' החוט.

- אם כל התהיליך מסתיים.

דוגמה נכתוב קוד שפותח חמייה חוטים. מה שהוא עונה זה לפתח חוט שידפיס עם מס' החוט שלו ויסיים.

```
#define NUM_THREADS 5

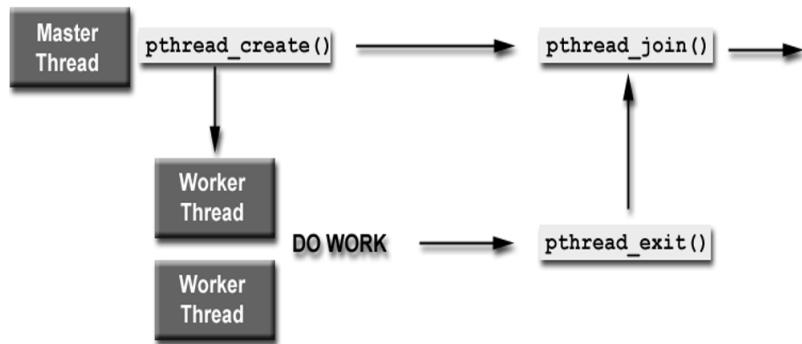
void *print_hello(void *arg) {
    int *index = arg;
    printf("\nThread %d: Hello World!\n", *index);
    return NULL; // Equivalent to calling pthread_exit(NULL);
}

int main(int argc, char *argv[]) {
    pthread_t threads[NUM_THREADS];
    int res, t;
    for (t = 0; t < NUM_THREADS; t++) {
        printf("Creating thread %d\n", t);
        res = pthread_create(&threads[t], NULL, print_hello, (void *) &t);
        if (res < 0) {
            printf("ERROR\n");
            exit(-1);
        }
    }
}
```

איור 24 : קוד לתוכנית שפותחת חוטים ולא עובד כמו שצריך

הבעיה עם הקוד הזה היא שהחוט של main יכול לסיים את הריצה לפני שהווטים האחרים יסיימו את הריצה שלהם ואז לא יודפס הכל כראוי. לשם כך נדרש להבטיח שמיין יסגר רק אחרי שכל השאר מסיימים.

הנוכחות של pthread_join מקבלת מס' חוט; ומוקומם לשמור בו את סטטוס היציאה של החוט והיא חוסמת את החוט הנוכחי עד שהחוט האחרון מסיים.



איור 25 : תרשימים זרימה של שימוש ב-join

הערה בחרזה לדוגמה הקודמת, עתה קטע הקוד החדש שייעבוד יהיה :

```

#define NUM_THREADS 5

void *print_hello(void *arg) {
    int *index = arg;
    printf("\nThread %d: Hello World!\n", *index);
    return NULL; // Equivalent to calling pthread_exit(NULL);
}

int main(int argc, char *argv[]) {
    pthread_t threads[NUM_THREADS];
    int res, t;
    for (t = 0; t < NUM_THREADS; t++) {
        printf("Creating thread %d\n", t);
        res = pthread_create(&threads[t], NULL, print_hello, (void *) &t);
        if (res < 0) {
            printf("ERROR\n");
            exit(-1);
        }
    }
    for(t=0; t<NUM_THREADS; t++) {
        res = pthread_join(threads[t], (void **) &status);
        if (res<0) {
            printf("ERROR\n");
            exit(-1);
        }
        printf("Completed join with thread %d status= %d\n",
               t, *status);
    }
}

```

איור 26 : קטע קוד לפיתוח חוטים עם join

חזרנו על [דוגמאות ל-Race Conditions](#)

נרצה בהינתן n תהליכיים, P_0, \dots, P_{n-1} , בלי אף הנחה אחרת על התזמון שלהם (הוא לא דטרמיניסטי), לפתור את בעיית קטע הקוד הקרייטי, ההנחה שליל (המסכם) היא שראית את ההרצתה ולא צריך לחזור על מה שנאמר שם.

באלג' של פיטרסון אנחנו מכנים שהתוור שלנו או שהאחר לא ירים את הדגל שלו. ההוכחה להתקדמות מאוד פשוטה: $tzut$ הוא או 0 או 1 ואז בכרח שאחד מה-while-ים לא מתקיים ובתווך אחד התהליכים נכנס לקרטע הקרייטי.

מיוטקס

נראה איך פרקטית משתמשים במשאב משותף. נשתמש באובייקט שנקרא mutex:

1. יוצרים את משתנה המיווטקס.

2. כמה חוטים מנסים לנעל (לקחת לעצם) את המיווטקס.

3. רק אחד מצליך והמיווטקס בבעלותו.

4. אותו אחד מרים את קטע הקוד הקרייטי.

5. הבעלים משחרר את המיווטקס ווחזרים לשלב 2.
6. משחררים את המיווטקס כشنגמרת הריצה.

- אפשר לATCHל מיווטקס ב-`pthread` או סטיטית לערך דיפולטיבי כלשהו, או דינמית (במהלך הריצה) באמצעות `pthread_mutex_init` שמקבל את המיווטקס לפני שאוחחל ורכבי אטריבוטים דיפולטיבים ומחזיר את אותו המיווטקס.
- כדי להרושא מיווטקס קוראים ל-`pthread_mutex_destroy` שמקבל את המיווטקס ונפטר ממנו.
- כדי לנעל מיווטקס קוראים ל-`pthread_mutex_lock` שמקבל את המיווטקס ומנסה לקחת בעלות עליו ואם הוא לא מצליח הוא חוסם את עצמו עד שהמיווטקס משוחרר ע"י חוט אחר.
- כדי לשחרר אותו קוראים ל-`pthread_mutex_unlock` עם המיווטקס והוא משוחרר אותו אם החוט שקוראלו הוא הבעלים של המיווטקס וכך. הוא יחזיר שגיאה אם המיווטקס כבר שוחרר או אם המיווטקס ננען על ידי אחר.

דוגמה כדי לעשות אינקרמנט/דיקרמנט בשני חוטים שונים עם מיווטקסים, נכתוב את הקוד הבא:

Thread 1:	Thread 2:
<pre>// Locking two different resources lock(a_mutex); lock(b_mutex); a=b+a; b=b*a; unlock(b_mutex); unlock(a_mutex);</pre>	<pre>lock(b_mutex); lock(a_mutex); b=a+b; a=b*a; unlock(a_mutex); unlock(b_mutex);</pre>

איור 27 : קטע קוד שמוביל לדד-лок

הבעיה עוד לא נפתרה כי אם אנחנו רוצים לחתך שני משאבים, נוכל להיכנס לדד-лок. לדוגמה, בקוד הבא חוט 1 קיבל את הлок של `a`, יהיה קונטיקסט סוויז', חוט 2 קיבל את הлок של `b` וניכנס לדד-лок. בהמשך נלמד איך להתמודד איתם.

מונייטורים (משתנים מותניים)

מיוטקסים מבצעים סינכרון של חוטים מבחןת בעיית המניעה ההדדית. אבל יש עוד הרבה בעיות סינכרון שונות.

דוגמה בתרחיש Barrier, נרצה לחכות בחוט הנוכחי שלו עד שכל החוטים האחרים מגיעים לנקודה מסוימת בריצה שלהם.

דוגמה בתרחיש Reader-Writers צריך לטפל בקריאה וכתיבה של הקובץ ולודא שמי שקורא את התוכן הרלוונטי.

דוגמה בתרחיש Bounded-Buffer יש צורך שקורא תוכן מבادر כלשהו ויש יצרן שמוסיף תוכן לאותו הבادر והשאלה היא איך הוא יקרא את המידע הרלוונטי לו.

הערה את כל אלה אפשר למשה באמצעות מיווטקס אבל זה לא תמיד cocci אלגנטוי.

דוגמה נניח שחווט 1 צריך את המשתנה של חוווט 2. נוכל לכתוב את קטע הקוד הבא :

Thread 1:	Thread 2:
<pre>while (true) { pthread_mutex_lock(&mut_var); if (canUseVar) { // use var canUseVar = false; } pthread_mutex_unlock(&mut_var); }</pre>	<pre>pthread_mutex_lock(&mut_var); init(var); canUseVar = true; pthread_mutex_unlock(&mut_var);</pre>

איור 28 : קטע קוד להמתנה למשתנה מחוץ אחר

כאן חוווט 2 לוקח את מיווטקס, מחשב מה שהוא צריך, מסמן את הדגל שקובע האם המשתנה אוטח ולשחרר את המיווטקס. חוווט 1 לוקח את הלוק, כשהוא מקבל אותו הוא בודק אם דגל ה"מכונות" מורם, אם כן, עושה מה שהוא צריך ומוריד את הדגל, ואחרת ממשיך בלאה האינסופית. ה-if הפנימי מתיחס למקרה שבו חוווט 1 קיבל גישה למשתנה לפני חוווט 2.

הערה המימוש הנוכחי משתמש ב-Busy Waiting ולא אלגנטיבי.

מוניטורים מיועדים לפטור בעיה זו (של התנייה בין חוטים). לモוניטור שתוי פ' מרכזיות :

- Wait שמקבל מיווטקס ומשתנה ומשחרר את המיווטקס (החווט חייב להיות בעלותו) וחוסם את החוווט עד להודעה חדשה, כשהוא מתעורר הוא ממחה עד שהוא יכול לחתום את המיווטקס.
- Signal שמקבל משתנה ומשחרר חוווט כלשהו שנחסמ ע"י Wait על אותו המשתנה ונוטן לו את המיווטקס.

דוגמה בחרזה לדוגמה הקודמת, נוכל למשוך זאת באמצעות מיווטקס באופן אלגנטיבי הרבה יותר.

Thread 1:	Thread 2:
<pre>pthread_mutex_lock(&mut_var); if (canUseVar) { pthread_cond_wait(&cv_var, &mut_var); } // use var canUseVar = false; pthread_mutex_unlock(&mut_var);</pre>	<pre>pthread_mutex_lock(&mut_var); init(var); canUseVar = true; pthread_cond_broadcast(&cv_var); pthread_mutex_unlock(&mut_var);</pre>

איור 29 : קטע קוד להמתנה למשתנה מחוץ אחר עם מוניטור

מה שקרה כאן הוא הדבר הבא : חוווט 1 לוקח את המיווטקס, אם הערך במשתנה עוד לא מוכן הוא חוסם את עצמו עד שימושים אותו שהוא מוכן, פועל עליו, ומשחרר את המיווטקס (ה-if אמרו להיות אם canUseVar לא דלוק, זו טעות).

חווט 2 לוקח את המיויטקס, שם את הערך במשתנה, מסמן שהוא מוכן לשימוש ומעיר חוט שצורך את המשתנה זהה ומשחרר מהמיוטקס.

אין לי כוח להקליד את כל אלה מחדש, אז הנה הפקודות הרלוונטיות למשתנים מותניים :

```
pthread_cond_init(pthread_cond_t *cv, NULL);
pthread_cond_destroy(pthread_cond_t *cv);
pthread_cond_wait(pthread_cond_t *cv,
                  pthread_mutex_t *mutex);
pthread_cond_signal(pthread_cond_t *cv);
pthread_cond_broadcast(pthread_cond_t *cv)
```

איור 30 : פקודות pthread-ב-למונייטוריים

על כל הפ' דיברנו כמעט ברודקאסט. ברודקאסט במקומות להעיר רק חוט אחד שירוחתי כמו `signal`, מעיר את כל החוטים שמחכים לערך המותנה.

מחסום

דוגמה נממש מחסום, ככלומר פ' שמחזירה רק כאשר שאר החוטים הגיעו לנקודת כלשהי.

```
#define NTHREADS 5
pthread_mutex_t *n_done_mutex;
pthread_cond_t *barrier_cv;
int n_done = 0;

main() { // Checking of return values omitted for brevity
    pthread_t tids[NTHREADS];
    int i;
    void *retval;

    pthread_mutex_init(&n_done_mutex, NULL);
    pthread_cond_init(&barrier_cv, NULL);

    for (i = 0; i < NTHREADS; i++)
        pthread_create(&tids[i], NULL, barrier, (void *) &i);
    for (i = 0; i < NTHREADS; i++)
        pthread_join(tids[i], &retval);
    printf("done\n");
}

void *barrier(void *arg) {
    // Checking of return values omitted for brevity
    int* id = (int *) arg;
    printf("Thread %d -- waiting for barrier\n", id);
    pthread_mutex_lock(n_done_mutex);
    ndone = ndone + 1;
    if (ndone < NTHREADS) {
        pthread_cond_wait(barrier_cv, n_done_mutex);
    }
    else {
        pthread_cond_broadcast(barrier_cv);
    }
    pthread_mutex_unlock(n_done_mutex);
    printf("Thread %d -- after barrier\n", id);
}
```

איור 31 : קטע קוד ליצירת מחסום

משמאלי אנחנו מתחילה הכל, יוצרים את כל החוטים עם פ' החוט שמוגדרת מימין ואוזעשים לכלום join. פ' המחסום מבקשת נעליה על מספר החוטים שישימו את עבודתם וועשה לו אינקרמנט.

אם זה לא החוט האחרון קוראים ל-wait על ה-CV של המחסום. אם זה כן האחרון עושים ברודקאסט ונונתנים לכלום לסיים לרווח. הארגומנט מסpter לחוט מה המספר שלו.

סיכום

סמאפור מאפשר לכמה חוטים לגשת למשאב משותף, ולא רק אחד כמו מיווטקס. לדוגמה, בכתיבת-קריאה לקובץ, אפשר שיחיה רק כותב אחד אבל הרבה קוראים. סמאפור מאפשר לתחילת עם כל מכסה של חוטים שרק נרצה.

בעת הכניסה לחלק הקרייטי נעשה דיקרמנט וכשנשיים נעשה אינקרמנט. חוט נחסמ אם הוא מנשה לעשות דיקרמנט והערך הוא 0 (כלומר אין עוד מקומות).

- כדי לאותחל סמאפור קוראים ל-`sem_init` שמקבלת אובייקט סמאפור ; האם לאאפשר שיתוף בין תהליכיים (לרוב 0) ; וערך התחלה למכסה.
- כדי לשיים שימוש בו קוראים ל-`sem_destroy` שמקבלת את הסמאפור ומהזירה 0 אם הצליחה ומספר שלילי אחרית (בדומה ל-`.(sem_init`)
- כדי לקבל גישה קוראים ל-`sem_wait` שמקבלת את הסמאפור ומהזירה עד שהייה אפשר לגשת אליו ואז מהזירה.
- כדי להחזיר גישה קוראים ל-`sem_post` שמקבלת את הסמאפור ועשה אינקרמנט לסופר הפנימי כדי לאפשר לאחרים לגשת אל המשאב.

זוגמה מה הבעיה בקוד הבא? אנחנו עושים אינקרמנט לאיונט מכל מיני חוטים שונים ויכול להיות שנעשה אינקרמנט באותו הזמן ונקבל .Race Condition

```
int counter;
void *foo(void * arg){
    for (int i = 0; i < 1000; ++i) {
        counter += 1;
    }
}

int main(int argc, char** argv)
{
    pthread_t threads[5];
    for (int i = 0; i < 5; ++i) {
        pthread_create(threads + i, NULL, foo, NULL);
    }

    for (int i = 0; i < 5; ++i) {
        pthread_join(threads[i], NULL);
    }
    printf("counter: %d\n", counter);
    return 0;
}
```

איור 32 : קטע קוד בעייתי

שם כך הומצא המשנה האוטומי, שעוטף טיפפ כלשהו ומאפשר לקרוא עליו `load`-`store` עם פרמטר כדי לקבל/לשים אותו "אוטומית" (מחזיק מיווטקס פנימי). ככה לא צריך להתעסק עם מיווטקסים והם ממומשים באופן פנימי (שתי הפ' מחוכות עד שهن מקבלות גישה למשנה).

שבוע VII | סמאפורים ומוניטורים

הרצאה

עוסוק עתה בפתרונות לביעיות תיאום שונות.

סמאפור הוא מחלוקת עם שני ערכיים: ערך, ורשימה של חוטים. יש לו שלוש פעולות בסיסיות (ראו איור) - העלת והורדת הערך ואתחול.

Down(S)	Up(S)	Init(S,v)
<pre>S.value= S.value - 1 if S.value < 0 then { add this thread to S.L; sleep();}</pre>	<pre>S.value= S.value + 1 if S.value≤0 then {remove a thread T from S.L; Wakeup(T);}</pre>	<pre>S.value= v</pre>

איור 33 : קטעי קוד עבור מימוש סמאפור

הערה את הפעולות אנחנו מבצעים בצורה אוטומית.

הורדת ערך מורידה את הערך (כל הבוד נמרוד) ואם "גמר המקום" בسمאפור, קרי הערך קטן מ於是, נשלח את החוט לישון עד להודעה חדשה ונוסיף אותו לרשימת החוטים שמחכים למקום בسمאפור.

באוטו האפונן העלאת ערך מעלה את הערך ומוציאו חוט מרשימה המתנה אם יש כזה.

הערה זה דומה למאחרת שימושת סמאפור בمساعدة שימושAIRה אנשים בהמתנה עד שמתפנה מקום.

הערה הרשימה היא לא בהכרח תור, אבל אם היא כן נאמר כי הסמאפור הוא הוגן.

בספרות האקדמית לעיתים מתיחסים להורדת והלאה C-P ו-V כי ככה זה בהולנדית.

דוגמה נציג מימוש של מניעה הדדית עם סמאפור. כל מה שצריך לעשות זה `mosop` בקטע הכניסה ו-`qu` בקטע היציאה כאשר הוא מאוחתל להיות 1. כל התכונות מתקיימות, כאשר היעדר הרעבה מתקיים רק אם הסמאפור הוגן, אחרת בשלושה חוטים יוכל 2 לשחק ביניהם עם הסמאפור והשלישי לעולם לא יקבל גישה למשאב. למעשה לעיתים מיותר מモושע ע"י סמאפור ביןארי (עם אתחול 1).

דוגמה נניח שאנו רוצים לבצע את משימה A לפני משימה B כהמשימות בשני חוטים שונים, כיצד נדרש זאת? עם מיטקס קלאסי זה קצת קשה כי אין פה קוד קרייטי, לכן נוכל להשתמש בסמאפורים. כל מה שצריך לעשות זה `qu` אחרי A ו-`down` לפני B עם סמאפור ביןארי או B יתחיל רק אחרי שהוא נכנס לסמאפור.

מגבלות הסמאפור

הערה סמאפורים לא פוטרים כל בעיה. נגיד אנחנו רוצים להעביר סף מ-A ל-B בחוט אחד ובוחוט אחר מ-B ל-A. נרצה להימנע מ-Race Condition. נוכל להזכיר על קטעי ההעברה קטיעים קרייטים ואז הכל יהיה בסדר.

דוגמה עכשו נניח שיש לנו חוטים שרוצים להעביר מ-A ל-B, מ-B ל-C ו-מ-C ל-D בהתאם. במקרה כזה, נוכל פשוט להזכיר על הכל קטיעים קרייטים אבל זה לא יעיל, כי אין לנו בעיה שהחוט הראשון והשלישי יעבדו בו זמנית כי אין קשר ביניהם.

נוכל במקומות לעשות סמאפור לכל חשבון ואז כדי להעביר בין שני חשבונות יצטרכו לחסום את כל שאר החוטים מלעת בחשבון בהם אנחנו עוסקים.

דוגמה אם היוו עושים העברה מעגלית (A ל-B ל-C ל-D) באربعة חוטים שונים) נוכל בקלה לקבל DD-lock כשהכל אחד ממחכה לחשבו אחד וחזריים לולאה.

הערה גם כישר לנו רק שתי העברות, חשוב לנעול את הסמאפורים לפי אותו הסדר תמיד כי אחרת יכול להיווצר DD-lock, כלומר צריך להיות סדר גלובלי למשאים.

בעיית הפילוסופים האוכלים

נניח שיש לנו שולחן עגול שבו יושבים פילוסופים שהם או אוכלים (למשך זמן סופי) או חושבים (יכולים להחליף בין המצבים). כשהם אוכלים הם צריכים לחתת שני צ'ופסטיקים וכשהם מסיימים לאכול הם מניחים אותם (משחררים אותם). שני פילוסופים לא יכולים להחזיק צ'ופסטיק יחד, כאשר בין כל שני פילוסופים יש צ'ופסטיק יחיד (כלומר כל אחד חוסם את שני שכניו).

נסמן $N - PN, \dots, P0$ פילוסופים ו- i צ'ופסטיק (משמעותו פילוסוף יונארו). לפני האכילה ישנה הפילוסוף $mown$ לסמאפור משמאלו ואז מימינו,أكل, ואז יעשה קפ לסמאפור משמאלו ואז מימינו ויחזור לחשוב וחזור חלילה.

זה לא יעבוד כי יכול להיות DD-lock - כל אחד לוקח את השמאלי וקורה קונטקטס סוויז' מיד אחרי ואז בסוף יצא שוכם מחזיקים את השמאלי אבל לא מצליחים לחתת את הימני.

הפתרון הוא לשבור את הסימטריה, לדוגמה באמצעות קביעה שאחד הפילוסופים יבחר הפוך. ככלומר, ככל ירימו את השמאלי ואז הימני אבל $1 - PN$ ירים קודם הימני ($[1 - N] [c]$) ורק אז השמאלי ($[0] [c]$).

משפט בפתרון הא-סימטרי הנ"ל לא ניתן להגיע למצב של DD-lock.

הוכחה: נניח בשלילה שהפילוסוף ה- i מרים צ'ופסטיק אבל אף אחד לא אוכל, ככלומר אף אחד לא הצליח להרים את הצ'ופסטיק השני שלו.

- אם $2 - N < i$: אם i מנסה לחתת את הצ'ופסטיק השמאלי שלו, $1 + i$ ולא מצליח, ולכן $1 + i$ הצליח לחתת את הצ'ופסטיק הימני שלו ולכך הוא אוכל סתיירה.

אם i לוקח את השמאלי שלו וUCCESSיו מנסה לחתת את הצ'ופסטיק הימני שלו, $1 - i$ ולא מצליח, ולכן מי שמיימנו לוקח לו אותו. אם מי שמיימנו לא אוכל זה אומר שמי שמיימנו ($2 - i$) לוקח לו אותו. אם הוא לא אוכל אז מי שמיימנו לוקח לו וכוכ' עד 0, שאם לוקח לו את הצ'ופסטיק שמיימנו זה אומר $1 - N$ לוקח אותו אבל ברגע שאצלו הסדר הפוך, זה אומר $1 - N$ מהזיך כבר בשני הצ'ופסטיקים והוא אוכל סתיירה.

- אם $2 - i = N$: אם $2 - N$ לא מצליח לחתת את האחד משמאלו זה אומר $1 - N$ מהזיך אותו ואז $1 - N$ אין את האחד שמשמאלו זה אומר $1 - 0$ יש אותו אבל זה הוא השני של 0 ולכך 0 אוכל.

אם $2 - N$ לוקח את השמאלי אבל ממחכה לימי המקרה זהה למקרה המתאים $2 - N < i$.

- אם $1 - N < i$: אם $1 - N$ מצליח לחתת את האחד מימינו אבל לא מצליח לחתת את האחד משמאלו (0) אז $1 - 0$ יש את שניהם ולכן הוא אוכל.

אם $1 - N$ מהזיך לאחד מימינו אז $2 - N$ לוקח לו ואם הוא לא אוכל $3 - N$ לוקח לו וכוכ' עד 0 אבל אם 0 לוקח מ-1 אז הוא בהכרח אוכל כי האחד מימינו לא נלקח כי $1 - N$ עדין מהזיך הראשון שלו.

הערה היעדר ההרעה לא טריומיאלי כאן והוא נראה לא מתקיים אם הסמאנט לא הוגן ואולי מתקיים אם הוא כן אבל צריך לבדוק את זה.

תנאי קופמן הכרחאים לדד-лок

1. לפחות משאב אחד צריך להינעל במצב בו לא משתפים את המשאבות (רק לתהילך אחד יש גישה אליו).
2. חוט אחד מחזיק במשaab אחד וambil שגישה למשaab אחר.
3. משaab יכול להשתחרר רק באמצעות שחזור ולונטרו של המחזיק בו.
4. המתנה מעגלית.

אפשר לתאר דד-лок ע"י גרפ' שבו הקודקודים הם חוטים ומשאים משותפים והצלעות הן ממשאב לחוט אם יש לו גישה אליו ומוחtot למשaab אם הוא ממחכה לו. אם יש מעגל בגרף כזו הרי שהתנאי הריבועי בתנאי קופמן מתקיים.

הבעיה עם שבירת הסימטריה הנ"ל היא שיכול להיות שرك אחד אוכל בכל פעם. אם נחלק את החוטים לזוגיים ואי-זוגיים ונגיד לכל אחד לנקחת משמאלו לימין והאחר מימינו לשמאלו נוכל להגיע למצב שלפחות 2 אוכלים ולא רק אחד. פתרונו זה נקרא פתרון LR. קשה מאוד לעלות על דד-ЛОקים כי זה תלוי בזמןו ולא תמיד קורה. כדי להימנע מdead-lockים, נגיד סדר על המשאים ונענול בסדר אחד ונשחרר בסדר הפוך תמיד.

יצrho-צרcn

חוט אחד או יותר יוצרים עבודה וחוט אחד או יותר צורכים עבודה (לדוגמא חוטי עבודה ומדפסות בהתאם). בעיות של דרישת יכולות לקרות גם בצד של העובדים, ככלומר שאחד ידרוס את העבודה של الآخر ויגרום לкриיסה של הכל, אבל גם מהצד של הצרכנים, למשל שעבודה תונפס פעמיים וכו' .

הערה אם הבאפר שעליו שמים משימות הוא בגודל חסום, נסתכל על הבאפר כציקלי, ככלומר אחראי שנגמר האינדקס האחורי חוזרים להתחלה.

במקרה של באפר ציקלי יש בעיות כבר עם צרכן וייצור אחד. אם הצרכן מתעכב על משימה אחת והיצור בנתים מייצר עוד ועוד משימות הוא יכול להגיע ולדרוס את המשימה בהתחלה שעדיין בפעולה ע"י הצרכן. בשלב כלשהו הצרכן והיצור מצביעים לאותו המיקום ואז נשאלת השאלה - האם הבאפר ריק או מלא? אי אפשר לדעת כי יכול להיות שהצרוך סיים הכל והגיע חוזרת לחזיות או שהיצור כתוב באפר שלם שהצרוך עוד לא עשה. אפשר להוסיף משתנה נוסף כמו משימות יש באפר ואז בזמנים שונים בין שני המצביעים הנ"ל.

Producer
(e.g., sending printing job)

```
while (COUNT==n);  
  
buffer [IN]=job;  
IN=IN+1 mod n;  
COUNT++;
```

Consumer
(e.g., printer)

```
while (COUNT==0);  
  
job=buffer [OUT];  
OUT=OUT+1 mod n;  
COUNT--;
```

איור 34 : מימוש של מצב צרכן-יצרן יחידים

למעשה הצרכן מחייב שהינה מה לעשות והיצרן מחייב שהיא מקום לכתב. חשוב להפוך את השורה של האינקרמנט ודקראמנט של הקאונטר לאוטומטית אחרת כਮון נקלט .Race Condition במקורה הכללי המצב קצר יותר מורכב, נוצר שולשה סמאפורים.

Producer
(e.g., sending printing job)

down(empty)
down(mutex)

```
buffer [IN]=job;  
IN=IN+1 mod n;
```

Consumer
(e.g., printer)

down(full)
down(mutex)

```
job=buffer [OUT];  
OUT=OUT+1 mod n;
```

up(mutex)
up(full)

Initial values:	
mutex	1
empty	n
full	0

up(mutex)
up(empty)

איור 35 : מימוש של מצב צרכנים-יצרנים רבים

כדי לכתוב לבסוף, יצרן צריך להחזיק במילוקס שהוא סמאפור בינארי. בנוסף, הוא צריך לחכות שהיה תא ריק, כלומר ש-empty (מאוחROL ל-0) לא יהיה שלילי. כשהוא מסיים, הוא מעלה את full, שמחזיק מיידע כמה תאים מלאים יש (מאוחROL ל-n). היצרן באותו הזמן צריך לחכות שהיה תא מלא, להחזיק במילוקס ואחריו זה להעלות את מספר התאים הריקים.

מונייטורים

דרך נוספת לפתרור את הבעיה זו היא עם מונייטורים. מונייטורים הם אובייקטים ברמות שפת התכנות שיעוטפים סכמת סנסרין כך שאפשר לגשת למידע רק באמצעות פ' ייעודיות. מונייטור פוטר אותנו מהתעסקות עם כמה סמאפורים.

מחליטים שלכל היותר חוט אחד יבצע פ' של האובייקט וככה מקבלים מניעה הדדית אוטומטית כי רק אחד יכול להגיע בכלל במשאב בכל רגע נתון. אם מישחו נסח' מנסה לגשת, הוא נחסם עד שמשחרר המשאב.

בדוגמה הזכרן-יצרן נגידיר פ' produce ו-consume שכותב/קורא ומעלה/מוריד קאונטר. כאמור אם מגיע לצרכן לפני שהוא יצרן אז הוא נחסם עד להודעה חדשה עד שmagiu יצרן ואנחנו בבעיה. כדי לפטור את זה יש משתנים מותניים, שלהם יש שלוש פעולות.

- wait שימושר את המוניטור וממחה שימושו עיר אותו אחריו שינה את הערך של המשתנה.
- signal שימושר חוט אחד שמחה (ברישימת המתנה שהיא תור).
- broadcast שימושר את כל החוטים שמחכים.

הערה בקורס הזה אנחנו מניחים כלל שתתהליכים מתקשרים באמצעות משאבים מסווגים (זכרו משותף). אפשר במקרה לתקשר באמצעות שליחת הודעות עם סוקטים או עלות בעיות דומות ועסקים בזה יותר בקורס אלג' מבוזרים.

תרגול

بعיית ה-Bounded-Buffer

כבר דיברנו עליו בהרצאה, זו בעיית היצרן-צרכן - נרצה שצרכן לא יקרא אם אין מה לקרוא והיצרן לא יכתוב אם אין מקום. הפתרון הוא סמאפורים סופרים.

mutex שמאוחחל ל-1 שדואג למניעה הדדית.

fillCount שסופר את מספר התאים המלאים.

emptyCount שסופר את מספר התאים הריקים.

Producer:

```
while (true) {
    produce an item
    down (emptyCount);
    down (mutex);
    add the item to the
    buffer
    up (mutex);
    up (fillCount);
}
```

Consumer:

```
while (true) {
    down (fillCount);
    down (mutex);
    remove an item from
    buffer
    up (mutex);
    up (emptyCount);
    consume the item
}
```

איור 36 : מימוש של בעיית הבאפר החסום

למה צריך גם אחד שסופר כמה מלאים ואחד שסופר כמה ריקים יש? נניח שנורוק את fillCount או יכול להיות שנוצרו הרבה יותר מאשר אפשר.

בעיית הפילוסופים האוכליים

ראינו את הבעיה כבר בהרצאה, נדנו בפתרונות לדד-ЛОקים שנוצרים מאלג' סימטריים (כל חוט מרים את אותו הקוד). למעשה, אף פתרון סימטרי לא חף מdad-LOCKים.

אפשר להוציא חוט נוסף שישמש מלצר/בורר שיחליט מי עושה מהותי. לחולפני אפשר לשבור את הסימטריה באמצעות פתרון LR. או שאפשר לעשות באקראי את סדר המשאבים.

אלג' למון-רבין

```
repeat
if coinflip() == 0 then           // randomly decide on a first chopstick
    first = left
else
    first = right
end if
wait until chopstick[first] == false
chopstick[first] = true          // wait until it is available
if chopstick[~first] == false then // if second chopstick is available
    chopstick[~first] = true      // take it
    break
else
    chopstick[first] = false     // otherwise drop first chopstick
end if
end repeat
eat
chopstick[left] = false
chopstick[right] = false
```

איור 37 : מימוש של אלג' למון-רבין

אנחנו קודם קובעים האם אנחנו מתחילה משמאלי לימיין או ימיון לשמאלי. לאחר מכן אנחנו מחליטים שהראשון ישחרר (ערך false במערך הוא פנו), נועלים אותו ומוחכים לשני. אם אנחנו לא מצליחים לקחת את השני ניכנס שוב ללולאה עד שכן יוכל לקחת אותו. לאחר מכן נבצע את ה"אכילה", וזו נשחרר את הcz'ופסטיקים.

הערה זה פתרון שימוש ב-`wait busy`.

בעית הקוראים-כותבים

נניח שיש לנו מבנה נתונים מסוון למספר חוטים שונים. יש חוטים קוראים שרק קוראים ולא כותבים ויש כותבים שיכולים לכתוב ולקרוא. נרצה לאפשר כמה קוראים בו זמנית, כותב אחד בכל רגע נתון וקריאה נחסמת בזמן שימושו כותב.

1. נשמר מספר (לא סמאפור) `readCount` שמאותחל ל-0 סמאפור ביןארו שמן על `readCount_mutex` ו-`writeCount_mutex`. גם סמאפור ביןארו ששומר את מספר הקוראים.

```

while (true) {
    down (readCount_mutex );
    readCount++;
    if (readCount == 1)
        down (write); // lock from writers
    up (readCount_mutex )
}

reading is performed // CS

down (readCount_mutex );
readCount - ;
if (readCount == 0)
    up (write);
up (readCount_mutex );
}

while (true) {
    down (write) ;
    writing is performed
    up (write) ;
}

```

איור 38 : מימוש הפתרון הראשון, משמאלי קוד הכותב ומימין קוד הקורא

הכותב מחייב שיכל לכתוב. הקורא חוסם כתיבה, קורא ואז אם הוא הקורא האחרון הוא משחרר את הכותב.

הפתרון הזה עובד אבל הוא מוביל להרבה אם יש הרבה קוראים של התוכן וקצת כותבים.

2. נסיף עכשו סמאנפור read בינהרי שימנע מקוראים ל לקרוא בזמן שכותב רוצה ל כתוב, writeCount, משנתנה שמאתחל לאפס, מיווטקס

ש망ן עליו וסמאפור בינהרי queue.

```

while (true) {
    down (writeCount_mutex )
    writeCount++; //counts number of waiting writers
    if (writeCount ==1)
        down (read)
    up(writeCount_mutex)

    down (write) ;
    // writing is performed – one writer at a time
    up (write) ;

    down (writeCount_mutex)
    writeCount--;
    if (writeCount ==0)
        up (read)
    up (writeCount_mutex)
}
}

while (true) {
    down (queue)
    down (read)
    down (readCount_mutex) ;
    readCount ++ ;
    if (readCount == 1)
        down (write) ;
    up (readCount_mutex)
    up (read)
    up (queue)

    reading is performed

    down (readCount_mutex) ;
    readCount - - ;
    if (readCount == 0)
        up (write) ;
    up (readCount_mutex) ;
}

```

איור 39 : מימוש הפתרון השני, משמאלי קוד הכותב ומימין קוד הקורא

עכשו הכותב מעדכן את מספר הכותבים שמחכים, חוסם קוראים, לוקח את הסמאפור של הכתיבה, כותב, משחרר אותו וمعدכן את מספר הכותבים שוב.

הקורא צריך לקחת את ה-read, readCount של read, queue והמיוטקס, לעדכן את מספר הקוראים ולהחסם כותבים, לשחרר את כל ה-*n*, לקורוא, ולעדכן חזרה את מספר הקוראים עם המיווטקס.

ה-read חשוב כי הוא מאפשר לכותב לחסום קוראים. ה-eueueu חשוב כדי שהרבה קוראים לא יריעיבו כותב אחד.

שאלות מ מבחנים

1. האלג' בשאלת דומה לאלג' של פיטרסון. ההבדל היחיד הוא שבמוקום לתת את זכות הקדימה לתהיליך الآخر, לוקחים את התו לעצמנו. ברור שזה לא יעזור כי אם יש קונטיקסט סוויז' מיד אחרי השמות ה-mutual שני החוטים חושבים שהם בעלי זכות הקדימה ונכנסים ביחד לקטע הקרייתי. לכן א' לא נכון, ב' כן נכון. ג' כן נכון כי צריך להוסיף – 1 – ב – i = turn ו – ד' לא נכון כי אי אפשר.

3. כאשר מספר תהליכיים חולקים גישה לבני נתונים מסוימים, עלולות ל奏ן בעיות.

(a) להלן פתרון לביעית הקטע הקרייתי:

```
shared boolean flag[2] = {false};  
shared int turn = 0;  
do  
{  
    flag[i] = true;  
    turn = i;  
    while (flag[1-i] && turn==1-i);  
    critical section  
    flag[i] = false;  
    remainder section  
} while (true);  
{}
```

מה דעתך על המימוש הזה?

- א) הוא מבטיח את תוכנת המניעה ההדדית במערכת עם שני תהליכיים.
- ב) הוא לא מבטיח פתרון לביעית הקטע הקרייתי במערכת עם שני תהליכיים.
- ג) אפשר לפטור את ביעית הקטע הקרייתי לשני תהליכיים בהוספה תווים בודדים (יש להוסיף אותם (2 נק') אם לדעתך אפשר).
- ד) אפשר לפטור את ביעית הקטע הקרייתי לכל מספר של תהליכיים בשינוי פקודה אחת בתכנית.

איור 40 : שאלה מבחן

2. הבעה עם הקוד הבא היא שלא מעדכן אוטומית. לכן א' לא נכון כי אם יש קונטיקסט סוויז' בדיק לאחר האינקרמנט (ולא נכנסים ל-if) אז גם כותב וגם קורא נכנסים יחד.

גם ב' לא נכון, כי בתנאי המקרה הקודם נכנסים שני קוראים, שניהם משחררים את הסמאפורה אז יותר כותבים יכולים לקרוא.

(ב) להלן פתרון לביעית קוראים כותבים. מה דעתך על המימוש זהה?

```
semaphore wrt_lock = 1;           READER:
int rd_count = 0;                 rd_count++;
                                 if(rd_count==1)
                                 down (wrt_lock);
WRITER:                         do_read();
down(wrt_lock);                   rd_count--;
do_write();                       if(rd_count==0)
up(wrt_lock);                     up (wrt_lock);
```

- (3 נק) (א) הוא מבטיח את תוכנת המוניה החדדית בין קבוצת קוראים לקבוצת כותבים.
(2 נק) (ב) הוא מבטיח את תוכנת המוניה החדדית בין הכותבים.

איור 41 : הנחיות שאלת מבחן

3. נקבע עבור הסעיפים הבאים מתקיימת עבורה מניעה חדדית והתקדמות.

א. (5 נק) עבור קטע הקוד הבא (עבור תהליך עם מזהה id, כל תהליך יירץ את הקוד עם ערך ה-id). ציינו איזו מהתנונות הנ"ל לא ניתן להבטיח, והראו דוגמה הרצתה נגדית בה תוכנה זו אינה מתקיימת. המשתנים door ו-race הם משתנים גלובליים. door מאותחן לערך true ו-race לערך -1.

```
1. while (true) {
2.   race = id;
3.   if door == false
4.     continue;
5.   else {
6.     door = false;
7.     break;
  }
<critical section>
```

איור 42 : שאלת מבחן

לא מתקיימת מנעה חדדית כי כש-door הוא true שני חוטים יכולים להגיע ל-block הクリיטי. התקדמות כן מתקיימת.

ב. (5 נק') הוצע השניי הבא בקטע הקוד. ציינו איזו מהתכונות הנ"ל לא ניתן להבטיח, והראו דוגמת הרצתה נגדית בה תכונה זו אינה מתקינה.

```

1. while (true) {
2.   race = id;
3.   if door == false
4.     continue;
5.   else {
6.     door = false;
7.     if race == id
8.       break;
9.     else
10.      continue;
}
}
<critical section>
```

איור 43 : שאלת מבחן

אין התקדמות כי אם ראשון נכנס ויש קונטקט טוויז'ר לפני בדיקת ה-race וחותם נוסף מגיע עד ל-if ואז חוזרים לראשון או שנייהם הגיעו ל-continue כל הזמן. אין מניעה הדדיות כי מניעה טריויאלית אינה נחשבת מנעה הדדיות.

ג. (5 נק') הוצע שניי נוסף בקוד. ציינו איזו מהתכונות הנ"ל לא ניתן להבטיח, והראו דוגמת הרצתה נגדית בה תכונה זו אינה מתקינה. **המשתנה *chowhao* משתנה לקליל!**

```

1. race = id;
2. if door == false
3.   win=false;
4. else {
5.   door = false;
6.   if race == id
7.     win=true;
8.   else
9.     win=false;
}

10. if (win==false) {
11.   Run entry code for the bakery algorithm (as we learned in
class)
}

<critical section>
```

איור 44 : שאלת מבחן

אין מנעה הדדיות כי חוט ראשון יכול לזרוץ, לקבל `win=true` ולהיכנס בקטע הקוד הクリיטי. לאחר מכן, יגיע חוט נוסף, יקבל `win=false`, יכנס למפעיה ויעבור בקטע הקוד הクリיטי כי הוא ראשון בתור.

```

1. race = id;
2. if door == false
3.   win=false;
4. else {
5.   door = false;           המשתנה xid הוא משתנה לוקלי!
6.   if race == id
7.     win=true;
8.   else
9.     win=false;
}

10. if (win==false) {
11.   Run entry code for the bakery algorithm (as we learned in class)
12.   xid = 0;
}
13. else {
14.   xid = 1;
}
15. Run entry code for Peterson's algorithm (as we learned in class)
with
  xid value as the process id

<critical section>

```

איור 45 : שאלת ממבחן

המניעה החזדית תישמר כי רק אחד יוכל לחמוך מהמאפייה ואילו רק אחד יוצא בכל פעם מהמאפייה ואז עליהם אנחנו עושים פיטרסון של שני חוטים ואז הכל בסדר. יש התקדמות כי אף אחד לא יתקע אותן.

מתי הקוד בסעיף ה'יל עדייף על אלג' המאפייה הקלסי? אם זה תהליך היחיד והרבה חוטים עדיף להשתמש בו, אבל אחרת החישוב של המאפייה הוא יקר עם כמה תהליכיים ואז עדייף את האלג' הזה.

נבחן את הפתרון בהבטים הבאים:

1. מנינה הדות: רק לקוות אחד מקבל מנה (MRIIN את הקוד של getMeal) בו זמנית.
2. הפקודות: אם של לקוות שפטמוני למנה, או לקוות כלשהו יכול לקבל מנה ווואי.
3. ייעילות: השף לא מבזבז משאבי CPU אם אין לקוות ממתין למנה או מקבל מנה.
4. עמידה בתנאי התו הסגול: לכל היורר 20 לקוות מעתינים במסעדת. לquoות מוגדר כמתמן במספרה אם הוא ביצוע של אחד משורות הקוד המסתובנת בין -ל-start of waiting block -end of waiting block

תוכנות:

```
int sem_init(sem_t *sem, int pshared, unsigned int value);
מאתהילת את הסטטוס של השם sem בערך 0 או גולר מ-0 או הסטטוס משוחה לתהילים. אם
ההזרה הוויא 0 אם פעולה הדיליה ואחרת -1.
```

```
int sem_post(sem_t *sem);
```

שколоוה לפועלה up שנלמדה בכיתה

```
int sem_wait(sem_t *sem);
```

שколоוה לפועלה down שנלמדה בכיתה

המספריים אינם הוגנים אלא אם נכתב במלפורש שם הוגנים.

בשל מגבלות הקורונה, לצורך עמידה במו הסגול, מוצנית מכירה רק **-by** take away וכן יכולים רק 20 אנשים להמתין במסעדת. נתן כי במסעדת שף אחד.

השף משרות את הלוקוות אחד אחרי השני, כך שבעל פעם הוא מכין מנה עבור לקוות אחד בדוק. כשההלקוות מגיע למסעדת והשף עסוק בהכנה מנה אחרה, הוא ימתין אם במסעדת כל היורר 19 לקוות ממתינים (לא כולל הוא עצמו). אם יש כבר 20 לקוות ממתינים, הוא ייעזב את המסעדת ולא יקבל מנה.

עלויות המסעדת מונמשת בהתאם הבא: השף ממושך ע"י ת浩יך ייחד שMRIIN את הפונקציה chef ובכל לקוות ממומש ע"י ת浩יך, כך שככל אחדMRIIN את הפונקציה customer נסיטים להכין מנה של לקוחות (דהיינו מסיטים את הפונקציה (prepareMeal(). הלקוח לוקח את המנה (הfonkziaה getMeal() והשף בוחר לקוות ממתין כדי לטפל בחומרתו. הפונקציות getMeal ו-prepareMeal ממושכות ע"י כתיבה למקומות ווכם ביוירון, כאשר נתן שהfonkziaה getMeal מסרימה אך ורק לאחר שהקראייה המתאימה ל-prepareMeal הסתיימה.

איור 46 : תנאי השאלה מהמבחן

א. (6 נק') מוצע הפתרון הבא:

```

Initialization of shared variables (איתחול משתנים משותפים):
sem_init(customers, 1, 0); // initialized to 0, shared among processes
sem_init(chef, 1, 20); // initialized to 20, shared among processes

void chef (void) {
    while(TRUE) {
        sem_wait (&customers);
        prepareMeal();
        sem_post (&chef);
    }
}

void customer (void) {
    \ start of waiting block - תחילת קטע הקוד בו ההליכים ממתינים לשירות
    sem_post (&customers);
    sem_wait (&chef);
    \ end of waiting block - סיום - קטע הקוד בו ההליכים ממתינים לשירות
    getMeal();
}

```

איור 47 : שאלת מבחן

4. אין מניעה הדדית כי שני לקוחות יכולים לעשות post ואז לא מחכים ב-wait (יש 20, הם לא נחסמים) ומריצים את getMeal(). יש התקדמות כי אחרי כל מנה השף מעלה את הסמאפור והוא מאותחל ל-20. יש עילוות כי השף יתקע ב-wait אם אין אף אחד שරוצה ממנה. אין עמידה בתנאי התו הסגול כי הלוקה ה-21 אמנים לכך לסמאפור השף אבל הוא יעשה post ובשלב זה הוא כבר בתוך קטע הקוד שנחשב המנתה.

```

void customer (void) {
    if (cnt_customers>=20) {
        // leave shop - ל跳出 חנות
        return;
    }
    cnt_customers = cnt_customers+1;
    \ start of waiting block - תחילת קטע הקוד בו ההליכים ממתינים לשירות
    sem_post (&customers);
    sem_wait (&chef);
    \ end of waiting block - סיום - קטע הקוד בו ההליכים ממתינים לשירות
    getMeal();
    cnt_customers = cnt_customers-1;
}

```

ב. (6 נק') לקוד התווסף שירות הקוד הבא (בהתגשה):

```

Initialization of shared variables (איתחול משתנים משותפים):
sem_init(customers, 1, 0); // initialized to 0, shared among processes
sem_init(chef, 1, 0); // initialized to 0, shared among processes
int cnt_customers=0;

```

איור 48 : שאלת מבחן, קוד השף לא השתנה

מניעה הדדית לא מתקינה מאותה הסיבה שהתקיימה בסעיף הקודם - לקוח אחד באמצעות קבלת המנה שלו אחראי שישים להתעס עם הסמאפורים וגם ללקוח נוסף והם שניהם לוקחים את המנה שלהם. התקדמות מתקינה כמו הפעם הקודמת. עילוות מתקיניות גם כמו הפעם הקודמת. התו הסגול לא מתקיים כי יכול להיותRace Condition.

בתרגיל נצטרך למשם את MapReduce: הקלט הוא סדרת איברים. שלב ה-map מפעילים פ' על כל האלמנטים באופן מקביל; בשלב הערבות/מיון האיברים ממונעים לסדרה חדשה; ובשלב ה-reduce מפעילים פ' שאוגרת ערכיהם.

דוגמה נספר את תזרויות האותיות בسطירינג. הקלט הוא סדרת מחרוזות. שלב ה-map סופר בכל מחרוזות כמה פעמים כל תו המופיע וככה מקבלים סדרה חדשה לכל מחרוזת. מסדרים מחדש את האיברים כך שלכל אות יש רשימה שמכילה את המחרוזות שמכילות אותה. בשלב ה-reduce פשוט סוכם כמה פעמים כל תו מופיע בכל המחרוזות.

שבוע VII | תזמון תהליכיים

הרצאה

אנחנו משתמשים במתזמנים כל פעם שיש משאב אחד והרבה משתמשים. אנחנו עוסקים רק במתזמן של ה-CPU.

דוגמא כמשמעותם ב-HDD נרצה לצמצם כמה שפחות תזוזה של הראש המגנטי שקורא מידע מתוך הדיסק, לשם כך נדרש לסדר את משימות הקריאה/כתיבה מתוך כמה שירות טוב.

ראינו את הדיאגרמה של המיצבים בהם יכולים להיות תהליכיים, ועתה עוסוק במתיה/איך בוחרים איזה תהליך רץ ובנוסף מתי אולי נדרש לבצע אותו לאחריו שהוא רץ יותר מדי (פעולות ה-schedule ו-preempt).

הערה כזכור אנחנו עוסקים במתזמן שהוא המוח של התזמון ולא ב-dispatcher שהוא טכנית מבצע את החלפה.

עבודה כלשהי מגיעה, מהכח שתוכל לרוץ (זמן החמתנה), ריצה (זמן ריצה) ומסימת. כל הפעולה הזו קורת במהלך זמן התגובה. זהה הפעולה של מה שקרה, כי כמו שראינו, אם ניגשים להתקן חיצוני לדוגמה, לא רצים עד שההתקן מחזיר תשובה. כמובן, יש לנו פריצים של ריצת CPU. לעומת זאת נזח את הבדיקה זו ונסתכל על ריצה לדבר רציף, כאשר העבודה שבאמצעו שלה לא רצים כי מחייבים למשהו נחשבת למעשה כמו בעבודות אחרות, כל אחת רציפה בפני עצמה.

המתזמן הוא בסה"כ אלג'. הקלט של הוא רשימת תכניות לתזמון; הפלט הוא לבחור את התכנית הבאה להרצאה; המטריה היא של מחשב יהיה "ביצועים טובים" (חוויות משתמש נעימה); הפעולות הנתונות לנו משתנות (האם אפשר לעזרה תהליכיים לפני שהוא מסתיים?).

מטרות התזמון

• זמן:

– מערכת תגובתית (זו חוות המשמש, אבל לא תמיד בשליטת המתזמן כי אחורי שהעבודה מתחילה לרוץ לא תמיד יוכל לעזרה אותה).

– ריצה שימושית מסוימת כמו שירות מהר (תוחלת מינימלית על התפלגות החמתנה).

– ריצה שהמערכת תאט כמה שפחות, ככלור לצמצם את זמן תגובה.

• גובה throughput, ככלור מספר העבודות שסיימו ליחידת זמן - זה רלוונטי לשרתוי web שבהם חוות המשמש לא משנה, אלא שכמה שירות מהר נבצע פעולות.

• ניצול מעבד גבוה, ככלור לצמצם את הזמן שהמעבד עסוק ולא בתקורה.

• הוגנות:

– כל עבודה מקבלת את חלקה הרואין לה (לא בהכרח שווה, אלא הוגן לפי איזושהי מדיניות).

– נמנע הרעהה של עבודות (אף עבודה לא תחכה אינסוף זמן).

– נתמוך בקיימות של משתמשים/עובדות (לדוגמה בשרת של האונ', ניתן עדיפות לפרו' על פני סטודנטים).

הערה המטרות האלה מוגבלות כשהמערכת עומסה מאוד ובהינתן סדר הגעה שונה של עבודות נוכל לקבל תוצאות מאוד שונות (אם הנו מסודרות בסדר עולה של אורך נוח לסדר אותן הרבה יותר).

הערה נניח לעתה שאין לנו קידמיות ושחקנים ההוגנים הם שווים, ובהמשך נעסוק במקרה שהזמן לא נכון.

יכולת להיות סתייה בין מטרות – כדי להגדיל את throughput-throughput (באמצעות סיום כל תהליך בלי לבזבז זמן על קונטיקסט סויצ'ים) נסתור את הרצון למזער זמן תגובה כי נאלץ לחכות שימושיות אחרות ישימו לפני שරץ חדשות.

אלגוריתמים Offline

הכוונה ב-*offline* היא שהאלג' מקבל את כל הקלט המלא ומוחזיר פلت בהतבסס עליו. online מקבל קלט בחלקים קטנים ומוחזיר פلت ש"נכון לאותו הרגע" כאשר התוצאה יכולה להשתנות בהינתן קלט חדש. אנחנו מניחים שיש לנו רישימה של עבודות לרבות זמן הריצה שלהם שאנו צריכים בסדר.

1. האלג' הכי פשוט שהוא offline הוא First Come First Serve, כלומר נחזיר את אותו הסדר בדיקוק.

זמן המתנה הממוצע יוצא כאן מאוד ארוך אם יש משימה גדולה מאוד בתחילת (בדומה לשירה שנוצרת מאחרוי משאית בכביש).

2. נסדר את העבודות לפי גודל עולחה אז לא ניתקע מאחרוי משימות ארוכות, זה נקרא Shortest Job First.

throughput-throughput ישר או הדבר אבל זמן המתנה ירד דרסטית. הבעיה כאן היא שם רישימת המשימות היא אינסופית (ב-*online*) נוכל לקבל הרעה, אבל זו לא בעיה כאן כי אנחנו מניחים רישימה סופית.

SJF מצמצם זמן המתנה ממוצע, ולמעשה הוא אופטימלי על המدى הזה. מאוד קל לחכיח את זה – אם יש סידור יותר טוב מ-SJF שהוא לא בסדר עולה של זמני ריצה, נחליף שני איברים שהם בסדר הפוך ונקבל סידור יותר אופטימלי סתייה.

גם *preemption* לא עוזר לנו כי אין לנו סיבה לשנות את דעתנו באמצעותם כי הכל ידוע לנו מראש (offline, כאמור).

מבחן זמן תגובה, זה גם אופטימלי, כי זמן התגובה מורכב מזמן הריצה (קבוע) עם זמן המתנה (אופטימלי).

אלגוריתמים Online עם זמני ריצה ובלתי preemption

נניח מודל פשוט, שלפיו העבודות מגיעות בזמן לא ידוע, כשהן כן מגיעות אנחנו יודעים את זמן הריצה שלהם, ואנו לא משתמשים ב-*preemption*.

1. האלג' הכי פשוט שהוא online גם כן הוא FCFS.

עקרון אנחנו שואלים בכל פעם שאנו מקבלים עבודה חדשה את המזמן מה הוא רוצה לעשות. במקרה הזה נגד לכולם לחכות עד שאלה שלפניהם יגמרו.

2. כדי לשפר את זמני המתנה והתגובה, יוכל להשתמש במתזמן SJF.

עקרון בכל פעם שנקבל משימה חדשה, נקבע שהבאה היא הקצרה ביותר מבין כל האלה שמחכות כרגע (ונזיו' אחרת ארוכות יותר).

לא נצורך מושימות (גם אם ארוכות) שרצות כרגע, אבל כשנתבקש לבחור אחת לrox, נבחר את הקצרה ביותר.

הערה אפשר להסתכל על זה בתור אלג' עם עדיפות כאשר בוחרים בכל פעם את העבודה עם העדיפות הגדולה ביותר, כאשר כאן העדיפות היא זמן הריצה (אחד חלקו למשעה).

גם FCFS הוא אלג' עדיפות כאשר העדיפות היא הזמן שהמתנו מזמן שהגענו.

חסרון עם SJF אונליין הוא כמובן ולכן לא יודע את העתיד, לשם כך נדרש דרך לשנות את דעתנו לפני סוף המשימה, קרי עם .preemption

אלגוריתמים Online עם זמני ריצה ועם **preemption**

עתה נניח שהעבודות מגיעות בזמן לא ידוע, כשהן מגיעות אנחנו יודעים את זמן הריצה שלהם, ואני כן יכולים להשתמש ב-preemption. המחיר של תוספת זו היא שקונטקטט סוויצ' עליה בתקורה.

1. נכנס בכל פעם את התהליך שזמן הריצה שנותר לו הוא הקצר ביותר, Shorest Remaining Processing Time.

עקרון בכל פעם שמדובר משימה חדשה והמתזמן מתבקש לבצע קבועה, נחליף לשימוש שנותרה לה הכי קצר זמן (או נשאיר אותו הדבר אם המשימה הנוכחית היא הקצרה ביותר).

המודל השלישי שלנו והוא הריאלי, הוא שהמשימות מגיעות בזמן לא ידוע ולא ידוע לנו מה זמן הריצה שלהם, אבל כן יש לנו .preemption. הבעיה כאן היא שאנו לא יודעים את העתיד - כמה זמן משימה תיקח, מתי לעצור מושימות ארוכות (אנו לא יודעים מי ארוכה) ולכן משימות קצרות נתקעות מאחוריהם ארוכות.

אלגוריתמים Online בלי זמני ריצה ועם **preemption**

1. יוכל לשער את הזמן שיקח לתהליך לrox (כאשר כאן אנחנו כן מתייחסים לתהליך בסדרת פרצי מעבד עם הפסקות O/I באמצע) ואז להריץ אלג' שכבר ראיינו, שמניחים שאנו יודעים את זמן הריצה.

עקרון כדי לחשב את זמן הריצה המשוערך, נחשב τ_n כאשר $\tau_{n+1} = \alpha t_n + (1 - \alpha)$ הוא משקלת ב- $[0, 1]$ ו- t_n הוא הזמן האמיתי שמדדנו.

חסרון שיטה זו לא תמיד נותנת תוצאות טובות מדי והיא דורשת הרבה מאוד תקורה.

2. נניח שאפשר להריץ כמה עבודות ביחד.

עקרון נגד למשימות לrox $\frac{1}{k}$ זמן אם יש k משימות, כלומר אנחנו מבצעים שיתוף מעבד.

כלל שיש יותר משימות נבעוד יותר לאט. בכל פעם שהמתזמן נשאל מה לעשות, הוא יגיד לדוחף גם את המשימה החדשה אל תוך המעבד (וכך נאט גם את המשימות הקודמות).

זמן ההמתנה הוא 0 וזמן התגובה הוא לא משלחו.

משימות קצרות לא נתקעות יותר מדי זמן, אבל הרבה משימות יחד יהיו איטיות. הביצועים של שיטה זו תלויים בסטטיסטייה של ה-workload - עם אילו משימות אנחנו עובדים. אם המשימות קצרות ברובן, אנחנו למעשה משערכים SRPT.

חסרון אם הרבה משימות באותו האורך ויחסית ארוכות, אנחנו מקבלים ביצועים יותר גורועים מהאלג' הנאבי, FCFS.

ביצועים מבחינת התאוריה, ניתן להוכיח שאם ההתפלגות מותה, כלומר שיש הרבה ערכים קטנים וקטנת גודלים, אז שיתוף מעבד הוא עיל. לעומת זאת, אם $\frac{\text{טטיית } CV}{\text{תוחלת}} > 1$ אז מקבל ביצועים טובים.

מחשבים מודרניים שומרים לוגים על תהליכי שב עבר ונוכל ללמוד על זמני הריצה ממש. מסתבר שמקבלים הסט' עם זנב פארטו, כלומר, $P(r) = \frac{1}{t} > t$ שזו דעיכה אקספוננציאלית ומאוד מותה.

המידע הזה לא תמיד מדויק כי זה משתנה לכל מערכת וכיו"ב, אבל בעיקרונו שיתוף מעבד זו שיטה טובה. הבעיה היא שאי אפשר לדוחף k משימות אל תוך מעבד, אלא רק 1.

Round Robin .3 הוא שערוך של שיתוף מעבד.

עקרון מריצים תחילה במשך קווואנטום זמן כלשהו (10 – 100 ms), עושים preempt ומוחזרים אותו בחזרה לתור.

למעשה אנחנו פותחים את האבסטורקציה של שיתוף מעבד וכל הזמן מחליפים מה אנחנו מריצים במקום להריץ הכל ביחד.

כל שהקוואנטום יותר קצר, זמן ההמתנה יותר קצר, אבל עם קוואנטום קצר יש יותר תקורה של סוויצ'ים. אם הקווואנטום מאד ארוך, התהליך יכול לסייע לפני הקווואנטום ונקבל FCFS. עם זאת, עבור אויזון כלשהו של קוונטים יחסית ארוך, נוכל לתת למשימות קצרות לרווח עד הסוף ומשימות ארוכות נרץ בחלוקת וכך נקבל את הטוב משני העולמות.

ביצועים פרקטית, אפליקציות כמו מדיה פליירים, תוכנות להורדת קבצים ואפליקציות יוטיליטי בסיסיות יקחו מעט מאוד זמן (כעשרה הדקות הפרומיל מקווואנטום באורך 100ms).

משימות שימוש פעולות בעיקר ע"י ה-CPU, למשל משימות חישוביות, לעומת זאת, מגיעות לעתים שכיחות הרבה יותר לתום הקווואנטום או לפחות חלק נכבד ממנו.

כדי למש RR, נשלח interrupt כל פרק זמן קבוע ומהשעון ובכל פעם שהוא מתקבל נרץ את האלג' של RR (זה בדיק מה שעשינו בתרגיל 2).

RR עובד בסביבת אונליין ומתמודד עם העבודה שהוא לא יודע מה הולך לקרות באמצעות preemption, והוא נותן יחס שווה לכלם. האם אפשר לעשות משהו יותר טוב מזה?

שבוע VII | תזמון עמוק

הרצאה

אנחנו דנים בנסיבות שmaguitות מתיישבו, מחייבות, רצות וויצאות מההטור. אנחנו לא מתייחסים ל-O/I אלא רק חישוב, ובתוכנות ריאקטיביות אחרי מאורע כלשהו יש פrz חישוב שהוא משימה.

ראיינו-sh SPRT (זמן נותר מינימלי) הוא הcyיעיל במצב אופלני ובמציאות RR הוא ממוש אונליין פרקט של שיתוף מעבדים, שהוא המקבילה האונליין של SPRT לצורך העניין.

כדי לשפר את התזמון, השתמש במידע מצטבר על משימות. בנגדו למה שעשינו עד כה, במקרה להתייחס על כל תח משימה כמשימה נפרדת, נסתכל על משימות חלק מתהילך כלשהו שנמשך גם אם הוא לא הגיע בתור. כאשרנו עושים *preemption* לשימה אחריו קווננטום, אנחנו יודעים שהוא לא קצר. המטרה שלנו היא להיפטר ממשימות קצרות כמה יותר מהר. לכן, נוכל להנמק את הקדימות של משימה שאנו שחייב לא קירה (סימנה קווננטום). לשם כך נשתמש בתור פידבק רב שלבי.

תור פידבק רב שלבי

תור פידבק רב שלבי הוא אלג' זמן לפי לאחר משימה מסוימת קווננטום, היא מועברת לתור אחר שבו שמיים משימות ארוכות. ניקח משימות מהטור המשני רק אם הטור המרכזי ריק. אפשר ליצור כמה תורים כאלה שלכל אחד מהם יהיה קווננטום אחר ואלג' זמן אחר.

דוגמא בניית ארבעה תורים עם קווננטומים, משמאלי לימיון, 80, 40, 20, 10 (הגדלה של הקווננטומים מצמצת תקופה כי ככל שעובר יותר זמן ניכר כי המשימה יותר ארוכה ואז לא כדאי לבזבז תקופה). בכל פעם אם המשימה לא נגמרה עד סוף הקווננטום מורידים אותה לתור הבא.

הערה היהת משימה כלשהי בתור מסוים מעידה על אורכה בערך (ככל שהוא יותר למטה הוא יותר ארוך).

לxicoms, משימה נכנסת בעדיפות הcy גבוהה, מרים את המשימה מהטור עם העדיפות הcy גבוהה ובתוך כל תור מותזנים לפי RR. האלג' מתעדף בסגנון SJF בלבד לדעת מה האורך שנוצר ומתי יוכל לשימושם בעזרת *preemption* והוא קליל מאוד ומשומש במידה צזו או אחרת ברוב המערכות המרכזיות.

פתרונות להרעה בתור פידבק רב שלבי

משימות ארוכות מורובות באופן די שכיח, נדרש למונעת זאת.

1. הקצתת זמן מעבד באופן ייחסי בהתאם לעדיפות (כמה שירות גבוה כמה שירות זמן) כאשר נממש זאת בתור הגרלת משימה להרצה עם הסת' בהתאם להקצתה. כך גם משימות בעדיפות נמוכות יקבלו יותר מ-0 זמן חישוב.

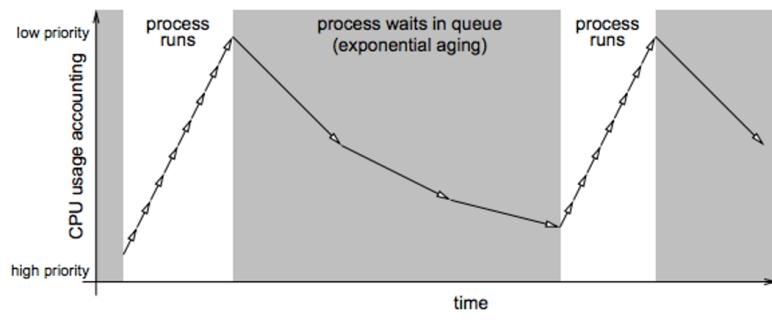
2. משוב שלילי - ככל שתהיליך רץ נוריד לו את העדיפות וככל שהוא ממתיין יותר נעלם לו את העדיפות.

דוגמא במערכות יוניקס המותזם הקלاسي הכיל 128 תורים עם 128 עדיפויות. עדיפויות 49-0 נשמרו לkernel וכל השאר למשתמשים. המתנה של kernel לדיסק (באמצעות *syscall*) הייתה בעדיפות 20 ולטרמינל בעדיפות 28.

העדיפות מחושבת ע"י: בסיס (50) + זמן החישוב עד כה (הבסיס הנוסף כדי שלא ניתן לאזרו של kernel).

קווננטום היה כל 10 תיקים, כאשר TICK של השעון קורה כל מאותייה. לאחר כל פrz (הקווננטום המלא או בלוק לפני) בודקים אם יש תהיליך בעדיפות גבוהה יותר, אם לא, עושים RR על התהליכים בעדיפות של התהיליך הנוכחי.

כל שנייה (100 תיקים) עוברים על כל זמני חישוב השמורים וחוצים אותם לשניים ובכך מעלים את העדיפות של colum. הפעולה זו גורמת לעלייה אקספוננציאלית בעדיפות (ראו גף).



איור 49: דוגמה לרכיבת המזמן הקלاسي של יוניקס

לחולופין, הרעבה היא לא בעיה אלא פיצ'ר. ככלומר, העובדה שימושיות קצרות מגיעה כל הזמן מעיד על כך שהמערכת בעומס יתר ולכן רק ברווח שאי אפשר להריץ הכל.

הערה הכוונה בעומס הוא היחס בין הקיבולת של המערכת לדרישות המשמש. נרצה שהמערכת תהיה לרובה-ב-100% עומס וכל דבר מעל זה לא יוכל להיות מושך כמו שצורך. מערכות ממשלתיות/גדולות נבחרות לפי העומס המצופה עליהם כך שלא אמרורים להגעה לעומס יתר.

למעשה, זה הגיוני להזכיר את הג'וב הארוך כי זה אומר שהוא לא רלוונטי למשתמש בטוחו הקצר וכן שימושה ארוכה אחת שköולה להרבה קצרות ועדיף להריץ הרבה מעט. לאחר שעומס יותר ירגע (לדוגמה בלילה), המשימות הארוכות יכולות לזרוץ ללא הרעבה.

בתוך פידבק רב שלבי, משימות חישוביות מקבלות עדיפות נמוכה בעוד משימות אינטראקטיביות (קצרות) מקבלות עדיפות מאוד גבוהה וירוצו מיד. בימינו יש משימות אינטראקטיביות ארוכות כגון רנדינג של תלת מימד ובמודל הנוכי הן לא יכולים לעמוד עדיפות גבוהה. לשם כך אפשר לתעדך לפי החלטו שכרגע המשמש מתממשק אליו (כך קורה בוינדוס).

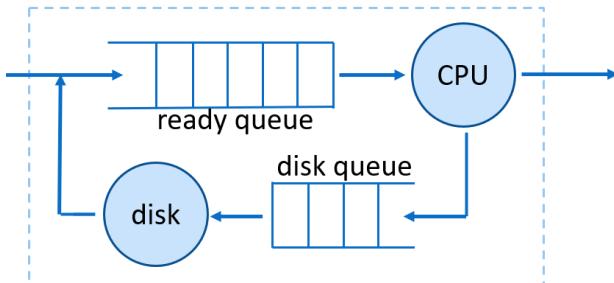
הערכת ביצועי מזמנים

בניגוד לניתוח באלג', נתעלם מהסבירויות של האlg' ויעניין אותנו רק איות הפלט שלו (נמדדוד לפי ממדדים כגון ניצולות וכו').

כיצד נעריך מזמן?

1. ניתוח אנליטי - הוכחות מתמטיות לביצועים.
2. סימולציה - להשתמש במימוש של המזמן עם קלט שאמור לייצג את המציאות.
3. מימוש - למשש את המזמן במערכת האמיתית ולראות אם הוא מביע היטב.

nbצע ניתוח אנליטי, ראו המודל שלנו. הפרמטרים של הניתוח הם ממוצע מספר המשימות שmagiuות ליחידת זמן, אלג' התזמון (FCFS) והזמן שלוקח לבצע כל משימה. נרצה לדעת מה זמן התגובה הממוצע.



איור 50 : המודל אליו נutowן לניתוח זמן

ניתוח תור 1/M/M

נניח כי יש לנו מערכת יחידה שבה העבודות מגיעה בקצב λ עבודות יחידת זמן בהפכים שמתפלגים פואסן (אחרי שהגיעה משימה הבאה מגיעה לאחר זמן שנקבע ע"י מ"מ פואסאו) עם פרטורה μ וכמות המשימות שהמעבד מבצע יחידת זמן היא אקספוננציאלית עם פרטורה (תוחלת) μ .

הערה אם $\mu > \lambda$ אז המערכת לא יציבה (אורך תור המוכנים ישאף לאינסוף) ולכון נדרש $\mu \leq \lambda$.

בحينו μ, λ , נרצה לדעת מה זמן התגובה הממוצע יהיה.

אם נניח שהפרש בין עבודות הוא קבוע וכל עבודה רצה זמן קבוע אז אם עבודה מגיעה כל 10 יחידות זמן וכל משימה היא באורך יחידת זמן 1 אז העומס יהיה 10% וכל מאד לעשות את זה לכל שני מספרים במקרה הדטרמיניסטי. אי אפשר להכליל כל כך בקהלות למקרה ההסת' כי אם אורך העבודה והפרש הזמן מאוד קרובים, לא תמיד תהיה הפרדה מלאה ביניהם.

משפט (משפט קטן) אם λ (קצב ההגעה) ו- μ (קצב העיבוד) או מתקיים $\bar{\lambda} = \bar{\mu}$ כאשר $\bar{\lambda}$ הוא זמן התגובה הממוצע ו- $\bar{\mu}$ הוא מספר המשימות במערכת בממוצע.

הערה נותר לנו למצוא את $\bar{\lambda}$ כפ' של λ ו- μ .

מצבים בה המערכת יכולה להיות

- אין תהליכי בכלל בתור.
- יש תהליך אחד בתור.
- יש 2 תהליכי בתור.
- ...

המעבר ממצב $i-1 + i$ קורה בקצב λ והמעבר $M-1 + i$ ל- i קורה בקצב μ . תרשימים מצבים כזה יוצר שרשרת מركוב. שרשרות מרכיבים לאורך זמן מתכנסות להסת' כלשהי למעבר בין מצבים (עם עוד כמה תנאים). נסמן ב- π ההסת' האסימפטוטית להיות במצב ה- i . באמצעות זרימה

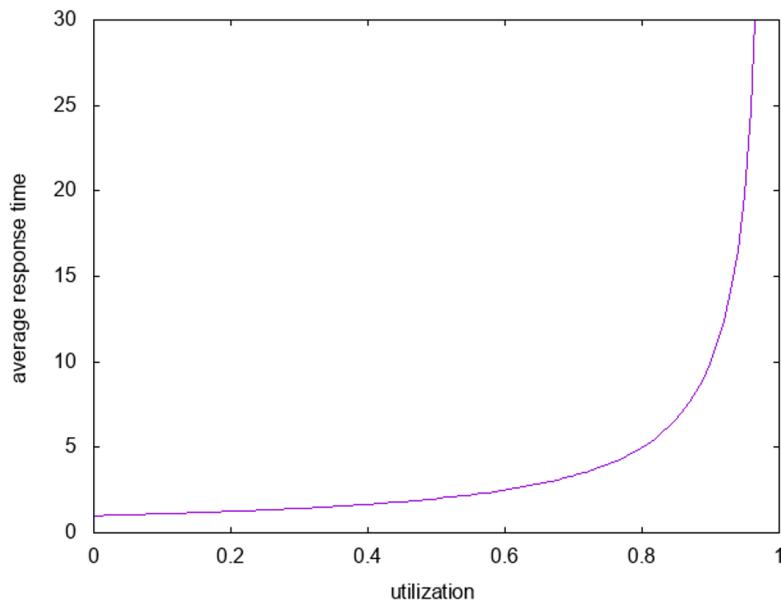
מאוזנת, כלומר שההסתט' למעבר מ מצב אחד לאחר שווה למעבר הפוך, מתקיים $\pi_1 \mu = \pi_0 \lambda$ ו לכן $\pi_1 = \frac{\lambda}{\mu} \pi_0$ ובאותו האופן $\pi_i = \left(\frac{\lambda}{\mu}\right)^i \pi_0 = \rho^i \pi_0$ ובמקרה הכללי $\pi_i = \frac{\lambda}{\mu} \pi_0$.

$$1 = \sum_{i=0}^{\infty} \pi_i = \pi_0 \sum_{i=0}^{\infty} \rho^i = \frac{\pi_0}{1 - \rho}$$

ולכן $(1 - \rho) \pi_i = \frac{\lambda}{\mu} \pi_0$. מתקיים

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

ובאמצעות משפט קطن נקבל $\bar{r} = \frac{\bar{n}}{\lambda} = \frac{\rho}{\lambda(1-\rho)} = \frac{1}{\mu(1-\rho)}$ שהוא אקספוננציאלי בኒצולת (ראו גוף).



איור 51 : שאיפה אסימפטוטית של זמן התגובה הממוצע כפ' של הኒצולת

הגרף הזה מלמד אותנו שגם במקרה הכללי, הרעבה אינה בינהית אלה תחילה שגדל עם העומס על המערכת.

בנוסף, אנו למדים כי חשוב שלא להתקר卜 ל-100% ניצול כי זה אומר שנקלט זמני תגובה מאוד גורועים, וכך נקנה מערכת עם קצר יותר כוח חישוב מהנדרש.

הערה כל זה הראננו רק עבור המודל הספציפי אבל טבעי כי זה נכון גם במקרה הכללי.

מערכות פתוחות וסגורות

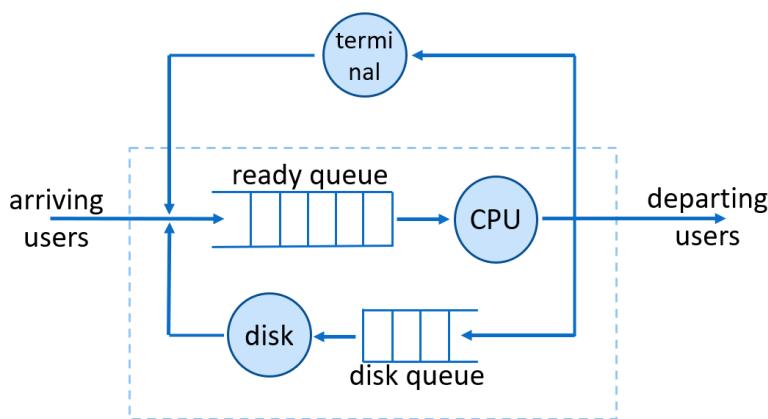
- **מערכת פתוחה :** אוכלוסייה אינסופית של עבודות שמגיעות למודל מבחוץ (מה שעכשו ראיינו).

ביצועי המערכת לא מושפעים על הגעת המשימות (אנשים ישלו בקשר לשרת בלי קשר לאיכות השירות). נהייה חייבם שהעומס יהיה פחות מ-100% ונמדד את איכות המתזמן לפי זמן התגובה.

- **מערכות סגורה :** מספר קבוע של משימות שחזרות על עצמן (לדוגמה מגן, צריך לבדוק כל כמה זמן האם הוא בטמפרטורה הנכונה, אם לא לעשות משהו זהה).

ביצועי המערכת נתונים ”פיבק“ למשימות (ביצוע איטי יאט את קצב ההגעה), העומס צריך להיות 100% ונמדד לפי תפקוד (כמה מהר מבצעים סיוב במערכת).

יש גם מערכות מעורבות גם פתוחות וגם סגורות (ראו איור). במערכת זו דומה משימות להיות משתמשים שבמציעים פעולה כלשהי שהיא או חישובית, או ממtingה לטרמינל/דיסק ואז חוזרות למערכת (סגורה) אבל יכולות להסתiens (פתוחה).



איור 52 : מערכת גם פתוחה וגם סגורה

ברגע שיש לנו רשת של תורים (גם דיסק וגם מעבד וכו'), חשוב לנו בעיקר מי צוואר הבקבוק.

דוגמא אם הפעולות הן CPU-Bound, המעבד הוא צוואר הבקבוק, אם זה הdisk הוא צוואר הבקבוק והמעבד לא עבד יותר מדי.

בכל מקרה, רק התזמון של צוואר הבקבוק הוא זה שמשנה.

תזמון בטוח אורך

עד כה בתזמון בטוח הקצר, ככלומר משימות שנמצאות בזכרון וכו', אבל יכול להיות שהמשימות שיש לנו דורשות יותר זיכרון משיש, לכן צריך לבחור אילו משימות שמים בזכרון וailו שומרים בדיסק לעתה. התקורה מאוד גבוהה לתזמון כזה ולכן יש לנו שני שיקולים מרכזיים:

להשאיר את המערכת אינטראקטיבית ; ולמצוא תערובת טובה של משימות שייאנו את השימוש במשאבים.

כלומר, לאון בין משימות CPU-Bound ו-I/O-Bound כדי שלא המעבד ולא הדיסק יהיו צוואר בקבוק.

תזמון הוגן

זמן יכול להתחשב גם בחקלים הוגנים (לא שווים) של תהליכי - לחת לכל תהליך את החלק ה"ראוי" לו, בהתחשב במסאים, דרישות, עדיפות פוליטיות וכו'.

1. **זמן זמן וירטואלי** : נבחר את התהיליך עם יחס הזמן המעבד שקיבל מול זמן המעבד ש מגיע לו (בהתחשב בשיקולים הנ"ל) ונ裏ץ אותו למשך קוונטים.

2. **זמן לוטו** : ניתן לתהליכי כרטיסי לוטו למשאים שונים וכמות הרכטיסים ייצגו את ההקצתה ש מגעה לתהיליך. בוחרים כרטיסים אקרים והטהיליך שיש לו אותו מקבל את המשאב הזה (המעבד, הדיסק וכו').

תרגול

קריטריונים מעשיים לבחירת שיטת זמן

- **niche utilization** - שהמעבד ירווח כמו שיטור (על מישימות אמיתיות, לא תקורה).
- **תפוקה throughput** - כמה שיטור מישימות מסיימות את הרצתן.
- **זמן המתנה waiting time** - כמה שפות זמן מהכנסה למתזמן ועד להתחלה הריצה.
- **זמן מוכלל turnaround time** - כמה שפות זמן מהכנסה למתזמן ועד סיום הריצה.

הערה יש הרבה דוגמאות שלא אוסף כי זה לוקח הרבה זמן ולא מלמד יותר מדי. הסטודנטית המשקיעה תמציא מושגיה התומסס דוגמה ותריץ את האלג' השונים. כל התרגול הוא תמצות מסודר של הרצאה.

שיטות זמן

1. **First Come, First Serve** .FCFS - מורים לפי הסדר.

דוגמה $(P_1, 10), (P_2, 1), (P_2, 1)$ (זמן המתנה, משימה) הזמן שייקח סה"כ הוא $2 \times 10 + 1 = 21$ (קונטסט סוויצ'ים).

הבעיה איתתו היא שהוא אמן הזמן אבל לא מתקף היבט במשימות ארוכות.

2. **Shorest-Job-First** .FCFS - מורים לפי זמן קצר ביותר (או פליין, preemption, בלי), אם יש שניים באותו אורך, עושים FCFS על המשימות השותג.

הבעיה איתתו היא שהרבה מאוד משימות קצרות מרuiboot משימה אחת ארוכה יותר, ובעיקר הוא לא פרקי כי אנחנו אף פעם לא באופליין.

דוגמה $(P_1, 8), (P_2, 4), (P_3, 9), (P_4, 5)$ (עם 7.5 זמן המתנה הוא סה"כ 7.5 דקות). לעומת זאת FCFS זמן המתנה זה עדיף כי שם זמן המתנה הוא 10.25 . הניצול מקסימלית, תפוקה זהה FCFS וזמן מוכלל קצר במעט מזה של FCFS.

.3 - Shortest Remaining Time First .(preemption).

יש לו את אותן היתרונות והחסרונות של SJF.

דוגמה : $(P_1, 8, 0), (P_2, 4, 1), (P_3, 9, 2), (P_4, 5, 3)$ כאשר הוסףנו עכשו זמני הגעה. נרץ אותם באופן הבא :

P_1	P_2	P_4	P_1	P_3
0	1	5	10	17

איור 53 : דוגמה לרכיבת מותגן SRRTF

זמן ההמתנה הוא $6.5 = ((10 - 1) + 0 + (17 - 2) + (5 - 3)) / 4$

preemption לא נספר זאת כהמתנה נוספת.

4. תזמון עדיפות - מרכיבים לפי עדיפות (לדוגמה טווח מספרים בהתבסס על תנאים כלשהם).

דוגמה FCFS זה תזמון עדיפות לפי זמן הגעה.

ניתן להגדיר עדיפות לפי סיבות פוליטיות או לפי דברים פנימיים (כמה זיכרונו זה דרוש). הבעיה עם השיטה זו היא הרעה של MERCHANTABILITY עם עדיפות נמוכה.

הפתרון לכך הוא aging - ככל שימושה ממתינה יותר זמן מעלים את העדיפות שלה.

5. Round-Robin - מרכיבים בקובאנטומים משימים.

אם המשימה מסוימת מוקדם היא תעוזב בפועל יוזם, אחרת נחייב אותה לעזוב את המעבד.

דוגמה עבר (P₁, 24), (P₂, 3), (P₃, 3) עם קוואנטום 4, קיבל את החלוקה הבאה :

P_1	P_2	P_3	P_1	P_1	P_1	P_1	P_1
0	4	7	10	14	18	22	26

איור 54 : דוגמה לרכיבת מותגן RR

הניצולות הנגעת במתגן זה בעיקר כי עושים הרבה קונטקט סוויצ'רים. הוא מאוד דומה ל-SJF, כאשר FCFS הוא עם זמן סיום הרבה יותר נמוך. כשחישנות גובה RR מבצע יותר טוב (גם במצבות).

6. Multilevel Queue - נחלק את התור לכמה תורים שונים, לכל אחד יהיה תזמון משלה ולתור הכללי יהיה גם תזמון כלשהו.

דוגמה תור עדיפות כללי שמכיל תורי RR לסטודנטים, פרופסורים וכו' בנפרד.

7. Multilevel Feedback Queue - תור רב שלבי כאשר MERCHANTABILITY יכולות לעבור בין תורים.

דוגמה הדוגמה הקודמת, כאשר MERCHANTABILITY שרצות יותר מדי זמן משתנות כדי לא לגרום להרעה.

אלג' כזה מוגדר ע"י כמה תורים ; מה אלג' התזמין של כל תור (כולל הכללי) ; באיזו שיטה קובעים לאיזה תור כניסה משימה כשהיא מגיעה ; متى משדרגים משימה ; متى משמנמקים משימה.

דוגמה Q_0 ו- Q_1 הם RR עם 8-16 מילישניות בהתאם ו- Q_2 הוא FCFS. כאשר חדש הוא מגיעו הוא נכנס ל- Q_0 , אם הוא מקבל 8 מילישניות מלאות הוא משונן ל- Q_2 , שם אם הוא מקבל 16 מילישניות מלאות הוא משונן ל- Q_2 .

תזמון במערכות מקביליות

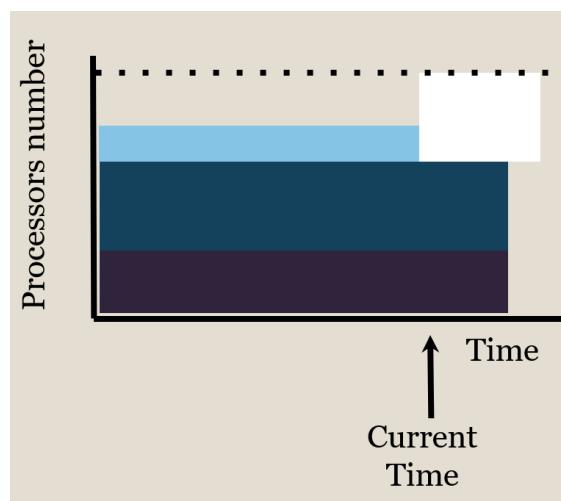
מחשיים על משמשים כדי לחשב דברים ממש מהר. לכל מחשי יש OS בסיסי ויש מחשי שמנהל איזו משימה יריד כל מחשי בתוך החוויה. המחשאים מרכיבים ביחס לשירותים כלשהו לא משתמשים ב-preemption אלא מוגבלים על ידי הדרישות שלהם (המשתמש מספק כמה זמן/זיכרון/מעבדים הוא רוצה).

AWS היא חברת הענן של AMAZON, ומספקת שירותים כזה בתשלומים (בamazon EC2), שקיים מאז 2006 והוא שלוט בשlish משוק הענן. בנוסף, 5% מכל האינטרנט עבר דרך AWS.

Akamai היא חברת החישוב המובילת בעולם (שאינה מוכרת), שהוקמה ב-1998 ע"י דניאל לוין שנרצה בפיתוח התואמים.

שיטות תזמון במערכות מבוזרות

1. FCFS - נריד משימות לפי הסדר, כאשר בכל פעם שמגיעה משימה חדשה (שדורש מספר כלשהו של מעבדים), נבדוק אם יש לנו מספר מעבדים פנויים גדול דיו, אם כן נכנס את המשימה, אחרת נגרום לה לחכות. ניתן להסתכל על זה בתור גרפף דו ממדי שבו ציר ה- x הוא הזמן וציר ה- y הוא מספר המעבדים ובזאת העבודה שיודיע לנו כמה זמן צריך לכל משימה, נוכל להרכיב את כל הבלוקים באופן אופטימי (בשלב הזה עדין FCFS אבל בהמשך יותר מתוחכם). ראו דוגמה לבניה כזו, כל מלבן הוא משימה אחרת.



איור 55 : דוגמה לריצת מזטמן FCFS במערכת מבוזרת

2. Backfilling - נריד משימות לפי הסדר ואם הגיעו משימה שאפשר לדחוף איפשהו בגרף נעשה זאת, גם אם היא לא ראשונה. הבעייה כאן היא שזה גורם להרעה של משימות שלא נדחפו במקומות.

3. EASY - נשמר רשימה של שימושות רצות וממתינות ונזכר כמה מעבדים הן דורשות ומהי הן אמורויות להסתאים, וכן מתי הן הגיעו (נמיין את הממתינות לפי פרמטר זה).

האלג' פועל באופן הבא :

(א) נתזמן את המשימות לפי FCFS.

(ב) נשרין מקום למשימה הראשונה שאינה אפשר להריץ.

(ג) נבצע Backfilling כמו שאפשר בהתחשב בשירויו. השלמה חורים אפשר לבצע בשתי דרכים.

- דחיפת המשימה לפי נתוניה המקוריים.

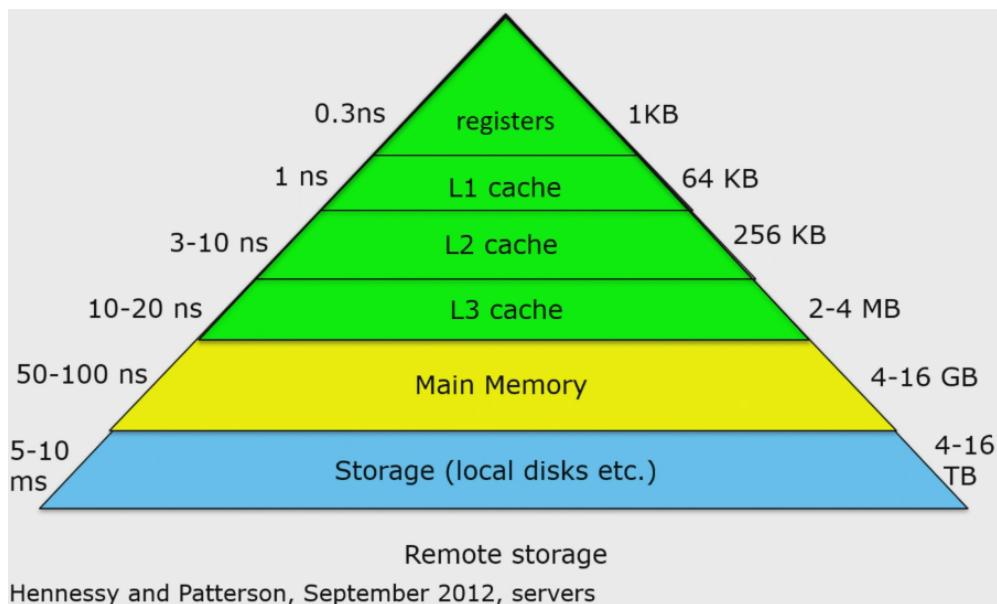
- מריחת המשימה עם פחות מעבדים אך לאורך יותר זמן.

זהו אלג' מאוד פופולרי ופשוט שמשיג ביצועים מאוד טובים.

שבוע VII | ניהול זיכרון

הרצאה

עד כה דנו באופן סכמטי בזכרון ועתה נעסק בו באופן יותר מפורט. הזכרון עצמו מחולק לשני סוגים זיכרון : הזיכרון המרכזי ומטמון (Cache). מטמון מחולק לשלושה שלבים - L1, L2, L3. לפי הסדר שלהם, מהירותם יורדת אבל מחירם וגודלם גם יורד (ראו איור).



איור 56 : חלוקה הזיכרון במחשב

שליש נאנו שנייה הוא בערך המהירות שבה עובד גם המעבד, לא רק L1 (שלושה גיגה הרץ זה פועלה כל... 0.000000000333 שניות). נשים לב שהגישה לדיסק היא איטית בכמה סדרי גודל (5) ביחס לזכרון המרכזי. כמובן, אם רגיסטר שקול להרים פריט שיש לידי על השולחן, הזכרון המרכזי שקול למשהו שנמצא ממני במרחק דקה וחצי והdisk שקול להזמנת משחו מודואר ישראל (שהוא גם מגע).

לאורך השנים, מהירות המעבד גדרה דרסטית (בהתאם לחוק מור, $1.5 \times$ בשנה) ואילו הזיכרון גדל ב- $1.07 \times$. לכן, יש פער מאוד גדול בין מהירות המעבד ל-DRAM (זיכרון דינמי, שהוא הזיכרון המרכזי). לשם כך כבר יחסית מוקדם הומצא המטמון, שמטרתו להיות מתוויז עם קיבולת קטנה אך מהירה בין המעבד לזכרון שומר דברים שהמעבד צריך. המטמון מבוסס על עקרון הлокאליות הזמןית והמרחבית.

דוגמה לוקאליות זמןית פרושה שאם ניגשנו לזכרון כלשהו, הסיכוי שניגש לזכרון הזה שוב 매우 גבוה.

אם אנחנו עושים אינקרמנט למשתנה, אנחנו קוראים וואז מיד כתובים, פעמיים קראנו לאותה הכתובת. לולאות בכך גם נחשבות לוקאליות זמןית כי הסיכוי להיות באותו מקום בקובד שוב בטוחה זמן קצר הוא גבוה.

דוגמה לוקאליות מרחבית פרושה שאם ניגשנו לזכרון כלשהו, סביר שניגש לכתובת סמוכה לו.

מערכות הם בבירור מקרה של לוקאליות מרחבית, וכך גם עצם העבודה שאנו מרכיבים פקודות באופן סדרתי (עד כדי קפיצות control flow).

העיקרון הוא לשמר את הדבר הבא שהמעבד סביר ביותר להיות צריך.

דוגמה כל שכבה מהויה מטמון של השלב שמעליה. אם קוראים משהו מהען אנחנו שמים אותו בדיסק, וואז בזיכרון, וואז בכל אחד מהקאשימים ורק אז ברגיסטרים של המעבד - כל אחד מהיר מהקדום.

טכנולוגיות זיכרון

1. מטמון משתמש בטכנולוגיית SRAM (סטטי) שהיא יקרה אבל מהירה, ושומרת את הזיכרון בו לתמיד כל עוד המחשב דולק, אבל כשהוא כבוי הכל נאבד (זיכרון נדיף, volatile).

2. הזיכרון המרכזי משתמש בטכנולוגיית DRAM (динמי) שהיא זולה אבל יחסית איטית ודורשת ריענון של הערכים בה כל כמה מיליאן פעימות (זיכרון נדיף).

3. דיסקים עובדים על משטחים מגנטיים שמסתובבים עם ראש שוזן וכל הפעולות האלה מכניות ולוקחות הרבה זמן (זיכרון לא נדיף).

לחופין יש SSD שהוא דומה ל-DRAM אבל יותר איטי ולצורך העניין דומה לדיסקים.

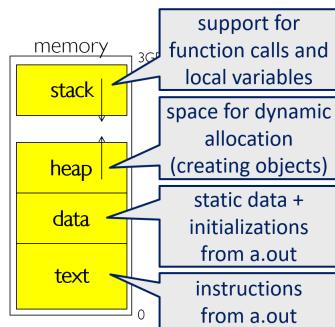
הערה כמשמעותו מרובות ליבת, הרגיסטרים ומטמוני L1, הם נפרדים לכל ליבה וכל מה שאחורי זה (L3, זיכרון וכו') הוא משותף לכל הליבות.

כל רכיב הוא תחת אחריות של דבר אחר - רגיסטרים מנוהלים ע"י הקומפיילר; מטמון ע"י חומרה; זיכרון ואחסון ע"י מערכת הפעלה. מערכת הפעלה מבצעת את האבසטרקציה של קבצים בדיסק, ניהול זיכרון בזכרון המרכזי וכו'.

מרחב הכתובות

מרחב הכתובות הוא כל הזיכרון שאליו יכול תחוליך לגשת. בארכיטקטורת 32-ビט אפשר לגשת ל-4 ג'יגה זיכרון ($4,000,000,000$).⁽²³²⁾ חלק מהזיכרון הזה שומר למערכת (בוינדוס שני ג'יגה למשתמש ובלינוקס שלושה ג'יגה נותרו למשתמש).

מרחב הזיכרון מחולק לאربعة סגמנטים שכבר דיברנו עליהם (ראו איור).



איור 57 : סגמנטי הזיכרון במחשב

יש פער בין מרחב הכתובות שתחוליך רואה לבין זה שיש במציאות (זוהי האבסטרקציה של מערכת הפעלה). בין שני ה"יקומים" האלה, צריך משחו שיתרגם כתובות. loader-ammo על תרגום הכתובות מהקוד המקורי למילוי הכתובות אבל עדין לא המציאות. נדרש עוד רכיב. הרכיב הזה נקרא MMU (Memory Management Unit) אחראי לפניו שהבקשה לזכרון נשלחת דרך ה- bus לתרגם את הכתובות. הוא מבצע תרגום מכתובות **לוגיות** לכתובות **פיזיות**.

שיטות תרגום כתובות

השיטה הכי פשוטה לתרגם היא הקצאה רציפה - נקצת לכל תחוליך אזור בתוך הזיכרון שיהיה רציף וכך לתרגם את x נחשב $x + base$ בזיכרון הפיזי. נשמר את $base$ והחסמ העליון של מרחב הכתובות של כל תחוליך ובכל קונטיקסט סוויצ' נחליף את המידע הזה בתואם לתחוליך. אם יש הרבה מאוד תהליכיים ויחסית מעט זיכרון (מה שקרה לרוב) אז אין מספיק מקום לכולם ואי אפשר למשתמש בשיטה זו בכלל.

```

1 physical(logical_x) {
2     return base + logical_x;
3 }
```

סגמנטציה

סה"כ הזיכרון שתחוליך משתמש בו הוא לרוב קטן הרבה יותר מאשר שהוא יכול לטכנית לגשת אליו, לכן נשמר זיכרון פיזי רק לסגמנטים שימושיים בהם.

כל סגמנט בפני עצמו הוא רציף וכך לתרגם את הכתובת x הווירטואלית בסגמנט כלשהו שמתחליל ב- $base$, שוב פשוט נחשב את $x + base$ בזיכרון הפיזי.

צריך לבדוק בנוסך שהזיכרון לא חורג מהחסים העליון על הסגמנט. זה בנויגוד להקצאה רציפה שבה מרחב הכתובות הוא כל הזיכרון שיש, ואז Ai אפשר פיזית לipyicr כתובות שחרוגת ממנה שאפשר לגשת אליו כי אין מספיק ביטים, אבל כאן בגלל שהסגמנטים קצריים יחסית אז כן אפשר.

טבלאות סגמנטים

בגלל שלכל תחлик יש כמה סגמנטים, נשמר אותם בטבלה שתכיל את כתובת הבסיס שלהם והגודל שלהם, כאשר כתובת לוגית עתה תכיל בביטים הראשונים את אינדקס הסegment ובשאר הביטים את הכתובת היחסית בתוך הסegment. בנוסף כדי שתהлик יוכל לשறין לעצמו מקום לסגמנטים בעtid (לדוגמה אם הוא רוצה ליצור חוטים ומחסנית לכל אחד מהם), נשמר גם בית ש Siegid לנו האם הסegment הנוכחי הוא בכלל חוקי (valid) או שהוא שמור להמשך, אם הוא לא, אסור לנו לגשת למקום זה זיכרנו.

```
1 physical(logical_x) {
2     seg_i = logical_x.segment // top bits of the address
3     segment = seg_table[seg_i]
4     if (segment.v == 0) {
5         throw exception;
6     }
7     if (logical_x.offset < segment.bound) {
8         return segment.base + logical_x.offset;
9     } else {
10        throw exception;
11    }
12 }
```

את הטבלה אנחנו שומרים ברגיסטר ה-Segment Table Address Register ה-STAB, שהוא שונה לכל תחליק. בקונטיקט סוייז' צריך להחליק את הרגיסטר הזה בהתאם לערכו בתהליק החדש. כך, סגמנטים של תחליק אחד אינם נגישים לתהליקים אחרים - זה מוסיף בטיחות זיכרון!

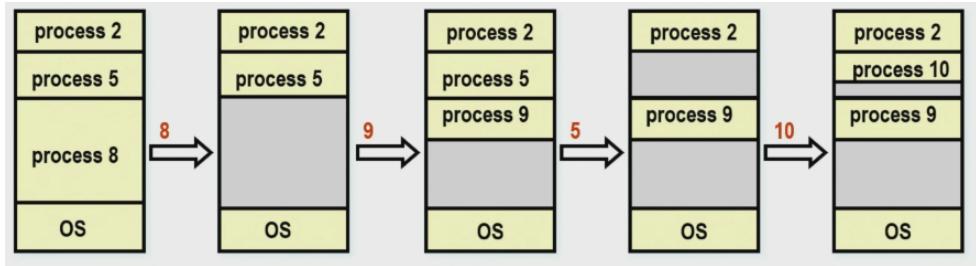
ברגע שריגיסטר ה-STAB מכיל את הפוינטר לטבלה של התהליק הנוכחי, כל שאר החישוב נעשה אוטומטיות ע"י החומרה (ה-MMU). המעבד יכול לבקש אותה כתובת וירטואלית ולקבל כתובות אחרות אם מדובר בתהליקים שונים (לכל אחד טבלה אחרת).

איפה החלק של מערכת הפעלה בכל הסיפור הזה? היא אחראית על המיפוי של הסגמנטים לזיכרון הפיזי וכן על שינוי ערך רגיסטר ה-STAB בהתאם להתקlik.

יש כמה שיקולים בהקצתה של סגמנטים - ההקצאות צריכה להיות רציפות, בגדים שונים, צריך לשמור מה הוקצה וכן לשחרר הקצאות.

פרגמנטציה

אם הסגמנטים שמוקצים הם בגדים שונים ובסדר מסוים, יוצא לפעמים שקטועים מוקצים גורמים לכך שהמקום אליו אפשר להקצות סגמנטים רציפים קטן או לא נוח, לדוגמה באירור הבא שבו לאחר הקצאות וסחרורים, יוצא שהמרחב בין ההקצתה של תהליק 9 ו-10 הוא בלתי שימושי בעליל. הבעיה הזו נקראת פרגמנטציה חיונית כי מוחץ להקצאות יש שביררים של קטועים שמקשים על הקצתה.



איור 58 : פרוגמנטציה חיצונית בזיכרון

הגדירה פרוגמנטציה היא מצב שבו יש מספיק זיכרון כדי להקצות סגנון לתהליך אבל לא מספיק רציף אלא מפוזר לאורך הזיכרון בקטעים קטנים.

הגדירה פרוגמנטציה פנימית היא המקרה שבו הקצאות יוצרת ממה שהוא צריך ואז נשאר מרוחח בפנים. פרוגמנטציה חיצונית היא שיש זיכרון חופשי בין הקצאות של תהליכיים.

הערה אנחנו ננסה לפתר רק את בעיית הפרוגמנטציה החיצונית.

אלגוריתמי הקצאות סגמנטיים

אלגוריתמים אלה מקבלים כקלט את רשיימת הקטעים החופשיים ובקשה להקצאה חדשה ופולטים באיזה אזור להקצות, ואיזה חלק ממנו לחקצוט.

1. First Fit - להקצות את המקום הראשון שפנוי. הסיבוכיות היא לנארית באורך הרשימה (נעוצר לכל היותר בסוף הרשימה).

לכואורה לשם ההקצאה אין חשיבות למיקום של רשיימת האזוריים הפנויים, אבל לשחרור זה כן חשוב כי אם היו לנו שני אזוריים פנויים שהופרדו ע"י קטע מוקצה שעכשיו שוחרר, נרצה לאחד את שלושת הקטעים לאחד גדול כדי לקבל תוצאות יותר טובות. לכן נニア שהרשימה ממויינת.

הבעיה עם האלג' היא שיכול להיות פרוגמנטציה בכתובות הנמוכות (כל הזמן נקצת ונשחרר גדלים שונים בהתחלה).

2. Next Fit - להקצות את המקום הראשון שפנוי אחרי ההקצאה הקודמת.

האלג' הזה מפזר את הפרוגמנטציה באופן יותר אחיד על הזיכרון כי לא נקצת ונשחרר כל הזמן בהתחלה ולא יקרה כלום בסוף אלא בפעם נדירות.

3. Best fit - להקצות את המקום שההקצאה בו היא הכי הדוקה (אין יותר מדי מקום חופשי לפרוגמנטציה).

כך אנחנו שומרים על אזוריים חופשיים יחסית גדולים ופרגמנטניים שלנו יחסית קטנים, זה חמור יותר כשהזיה מגע למצב זהה משפייע כי באמת אי אפשר להשתמש בזה.. עם זאת, נטרך לעבור על כל הרשימה כל פעם במקום לעצור מוקדם יותר.

4. Compaction - נזיז את החורים (את הזיכרון סביבו למשה) כדי שהאזורים הפנויים יהיו יותר ארוכים ונמצאים פרוגמנטציה.

הvisorון כאן הוא שהעתקה של זיכרון לוקחת הרבה זמן. בנוסך, המטען כבר מיותר כי אנחנו מזיזים לו את הכתובות והן כבר לא אוחת ליד השניה האחורי הזהה.

.5 - במקומות שסגןנט יהיה רציף, הוא יהיה עשוי מדפים בגודל קבוע, שכל אחד מהם יוקצה באופן בלתי תלוי.

Paging

נחלק את מרחב הכתובות לדפים בגודל קבוע (לרוב 4 קילו-בייט). נחלק ריעוניות גם את הזיכרון הפיזי למסגרות באותו הגודל של דפים ואז יוכל להתאים כל דף וירטואלי למסגרת פיזית. העתקה זו נעשית בידי OS וה-MMU משתמש בה כדי לגשת לזיכרון הפיזי.

בגלל שכל מסגרת היא בגודל קבוע, אף פעם אין פרגמנטציה. את ההעתקה מדפים למסגרות נשמר בטבלה (פרדרת לכל תחילה) שבה למעשה משתמש ה-MMU. האינדקס בטבלה הוא מספר הדף, הערך הוא אידנסט המסגרת שהותאמת לו.

כדי לתרגם כתובות לוגיות נחלק אותה לשני חלקים - ה-20 ביטים הראשונים יהיו אידנסט הדף וה-12 התוחתונים הם ההיסט בתוך הדף (12 ביטים זה בדיק מספיק ל-4 קילובייט).

נשמר בטבלה בית שאומר לנו האם הדף ממופף או לא. הפעם לא צריך לבדוק שההיסט נמצא בטווח כי הוא פיזית לא יכול לצאת מ-4 קילובייט.

```
1 physical(logical_x) {  
2     page_i = logical_x.page // top bits of the address  
3     page = page_table[page_i]  
4     if (page.v == 0) {  
5         throw exception;  
6     }  
7     return page.frame + logical_x.offset;  
8 }
```

הערה הסכמה של המסגרת עם ההיסט בתוכה היא למעשה הדבקה של ביטים נמוכים עם גבויים ואפילו לא סכימה של שני מספרים כי כאמור הביטים לכל דבר נפרדים. זה חוסך לנו כמה פעולות והופך את זה לאפילו יותר יעיל.

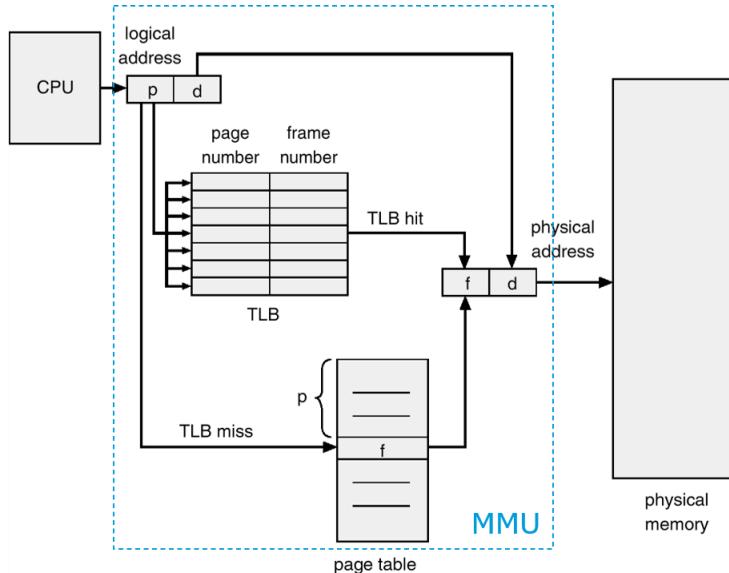
שבוע XII | דפדוֹן

הרצאה

החישוב של המסגרת הנדרשת לגישה לזיכרון בכל פעם נוספת תקורה, כדי להקטין אותה יש מטען ספציפי לטבלה הדפים, שנקרא ה-TLB (Translation Lookaside Buffer). בנוסף, טבלת הדפים עצמה לוקחת הרבה זיכרון. כדי להקטינה, נגדיל את גודל הדף (נבחר גודל אופטימלי שמאזן את הטריאידון).

טריאידון הוא זהה - מצד אחד דפים קטנים גורמים לפרגמנטציה פנימית קטנה יותר (חצי מהדף האחרון שמוקצת לטגןנט בתוחלת) אבל דורשים טבלת דפים גדולה.

נסמן ב- p את גודל הדף, s גודל התהיליך (בזיכרו), e גודל של כניסה בטבלת הדפים (מספר המסגרת ובייט ה- s). במקרה זהה התקורה היא $\frac{p}{2} e + \frac{s}{p} e$ (חצי השמאלי הוא גודל טבלת הדפסו הימני וזה הפוגטנטציה הפנימית על הדף האחרון). עם קצת חזו"א מגיעים לכך שהגדול האופטמלי הוא $\sqrt{2s \cdot e}$. עבור ערכים טיפוסיים בעבר $e = 64\text{bit}$, $s = 1\text{MB}$, $p = 4\text{KB}$ נקבל $\sqrt{2s \cdot e} = \sqrt{2 \cdot 1\text{MB} \cdot 64\text{bit}} = \sqrt{2 \cdot 1\text{MB} \cdot 64 \cdot 8\text{bytes}} = \sqrt{2 \cdot 1\text{MB} \cdot 512\text{bytes}} = \sqrt{2 \cdot 1\text{MB} \cdot 2^9\text{bytes}} = \sqrt{2 \cdot 2^{29}\text{bytes}} = \sqrt{2^{30}\text{bytes}} = 2^{15}\text{bytes} = 32\text{KB}$.



איור 59 : ארכיטקטורת תרגום כתובות עם ה-TLB

באյור ניתן לראות גם את ה-TLB. בכל פעם שאנו מתחבקשים לתרגום כתובות, נשווה באופן מקביל בין כל הערכים במתמונן, אם הדף שמשתמש בערך שמש, אחרת נחשב את הערך בטבלת הדפים הקלאסית ומשם נמשיך את החישוב המקורי. ב-TLB יש בערך 64 כניסה, כלומר העתקות של דפים למסגרות.

זיכרון וירטוואלי

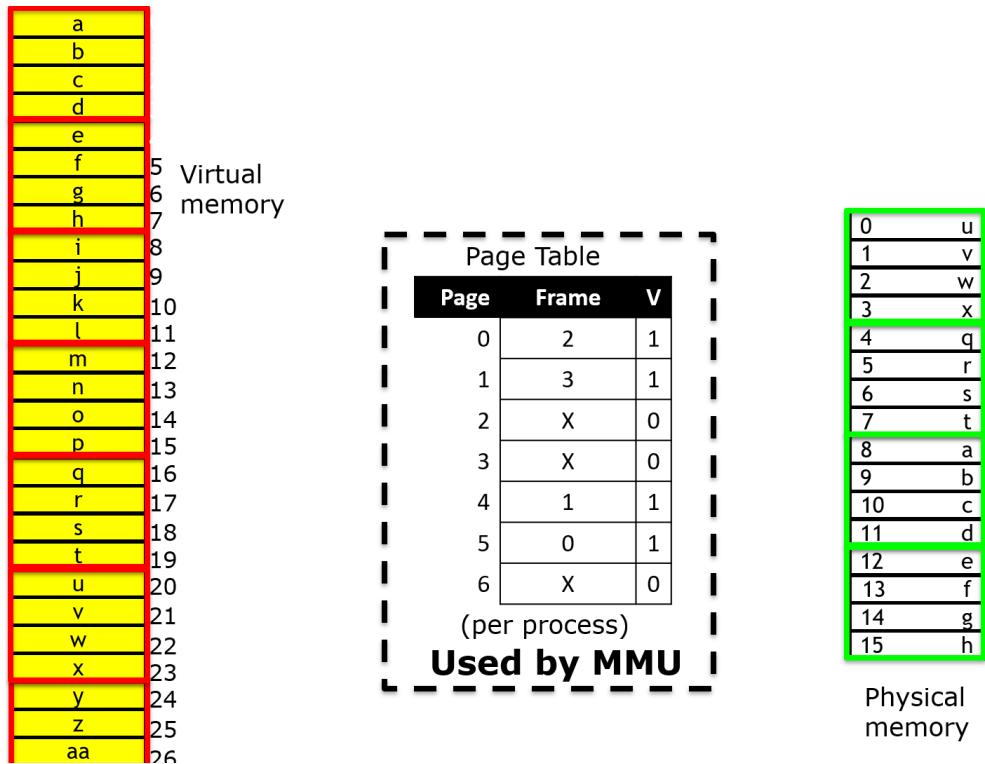
מספר התהיליכים בלתי מוגבל בעיקרו וכל תהיליך חשוב שהוא בלבד ושיש לו את כל הזיכרון (אולי חוץ מערכת ההפעלת) וכך יש הרבה יותר זיכרון לוגי מאשר פיזי. עם זאת, תכניות לא משתמשות בכל מרחב הכתובות שלחן כל הזמן, ולמעשה אין צורך למפות חלקים שלא משתמשים בהם לזכור פיזי.

דוגמא אם קוראים לפ' `mit` בהתחלה, אנחנו לא משתמשים בה בהמשך ולכן כל מבני הנתונים שהשתמשו בהם בתוכה לא ישומשו בהמשך התכנית. לחופין, קוד שמתפל בשגיאות מסוימות לעיתים נדירות יחסית ולאחר טיעינה שלו לזכור כל הזמן היא די מיותרת.

אי-הकצתה של חלקים לא מנוצלים (ושמירה שליהם בדיסק בנתים) עוזרת להורדת העומס על הזיכרון וגם מקלת על אתחול תהיליכים (לא צריך לטעון את הכל על ההתחלה).

דף לפי דרישת

כדי לספק את הווירטוואליות האמורה, משתמש בדף לפי דרישת, כלומר נביא דפים לזכור כנדרך אותם ונזכיר אותם לדיסק כשכבר לא.



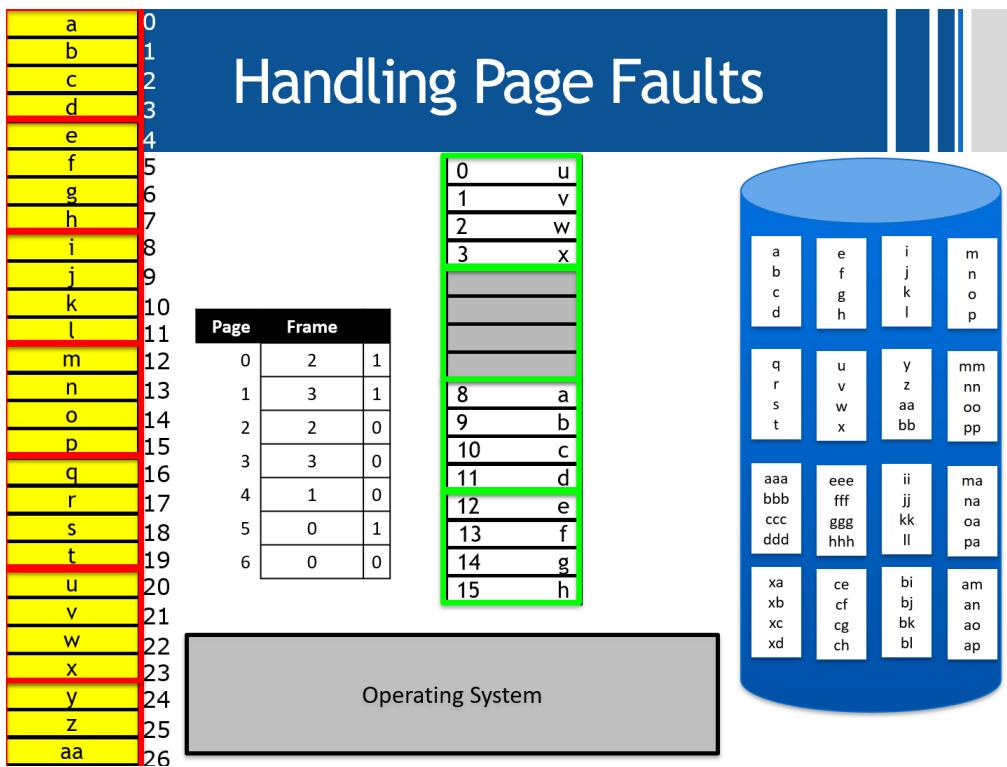
איור 60 : דוגמה לדפודוף לפי דרישת

באיור ניתן לראותות שחלק מהדפים בטבלה כרגע לא מוקצים, כי אין מספיק מקומות, ולכן הם נמצאים בדיסק ובית ה-*ה-υ* שלהם הוא 0.

גישה לזכרון לא ממופת

- המעבד לא יכול לגשת למידע זהה ולכן MMU יזרוק exception (שנקרא PAGE FAULT exception).
- ההאנדר של ה-*ה-υ* הזה נכנס לפועלה.
- ההאנדר מתחילה להביא את הדף מהדיסק.
- ה-OS מרדימה את התהיליך.
- כשהדיסק סיים להביא את הדף, הוא שולח interrupt שמעיר את התהיליך.
- התהיליך מבצע את אותה הפקודה שגרמה לו לשגיאה, שוב.

דוגמה נניח שאנו רוצים לבצע R1, 12 wן כולם לחת את הכתובת ה-12 לרגיסטר. הכתובת 12 היה בדף 3, שהוא לא ממופת בהתחלה, ולכן PAGE FAULT, נקרא להאנדר וכי', הדיסק יביא את הדף, יזרוק אינטראפט במסגרת סכמת ה-DMA שראינו (לפי הזיכרון יכול לעבוד ברקע ולספק בזמנים שהמעבד עושה דברים אחרים) ונמשיך ברגיל.



איור 61: דוגמה לגישה לדף לא ממופה

מה אם אין לנו מסגרת פנوية למפות דף אליה? לדוגמה אם יש לנו המון תהליכי שימושים בכמה דפים, יש לנו יותר זיכרון וירטוואלי מפיזי. נאלץ לזרוק דפים, ככלומר להעתיק את תוכנים חוזרת לדיסק לפנות את המסגרת שלהם לדף אחר. מי שנזורך נקרא הדף הקורבן.

כיצד נבחר את הקורבן?

אפשר לבחור אקראית, אבל אז זה יכול לזרוק החוצה דפים שמתמשכים בהם באופן שכיח (זו למעשה התופעה הכי גורעה שנרצה למנוע). אידאלית, אם יש לנו יכולת לראות את העתיד, נבחר את הדף שהשימוש הבא שלו הוא הכי רחוק (אלאג' זה נקרא האלאג' של בילדי). אבל כמובן שאין לנו יכולת לדעת את העתיד. למעשה השימוש המרכזי של האלאג' האופטימלי הוא להשוואה ביחס אחרים, שכן ניתנים למימוש האלאג' האקראי והאופטימלי הם הקצאות השונות של הסקלה (אחד לא משתמש בשום מידע, الآخر משתמש בכל המידע, אפילו כזה שאין). חשוב לבדוק כל פעם האם האלאג' חדש שאחננו מציעים הוא בכלל טוב יותר מאשר מקרארי, ואם לא אין מה להתייחס אליו. החלטה על הגירוש היא של מערכת הפעלה. היא יודעת אילו מיפויים קודמים היא עשתה.

אלגוריתמי גירוש

דוגמה FIFO - נשמר בראשימה מקוישת את הדפים שמייפינו ובכל פעם שנדרש לגרש דף, נגרש את הראשון שמיופה (הותיק ביותר).

חסרון יכול להיות שהדף הותיק ביותר ניגשים אליו באופן שכיח. בנוסף, האנומליה של בלאדי היא גם חסרון של FIFO. האנומליה אומרת שכורה נדמה שככל שיש יותר זיכרון יהיה פחות PAGE FAULT-ים. עם זאת, בלאדי ראה שזה לא המצב ב-FIFO, כלומר שהמצביע מחמיר עם יותר זיכרון.

בוינדוס NT השתמשו באlg' הזה כי הוא ב"ת בחומרה, שכן הוא משתמש רק במידע של מערכת הפעלה (ה-OS), אם אף אחד לא יספר לה, לא יודעת שום דבר על הזיכרון בלבד כшиб PAGE FAULT.

הערה כל מה שאנו מנסים לעשות בסופו של דבר זה לזהות את ה-Working Set של כל תהליך, כלומר הדפים שחשוב לשמור כי התחילה משמש בהם באופן שכיח והשאר שאפשר לזרוק. ההבנה הזו הושגה יחסית מאוחר במחקר על מערכות הפעלה (שנות ה-70). ה-Working Set עצמו הוא מבוסס על עקרון הלוקאליות - יש לנו אוסף סופי וקטן יחסית של דפים שנשתמש בהם שוב ושוב (כי כבר השתמשו בהם).

כיצד מוגדר ה-Working Set פורמלית?

הגדרה (הגישה הפרטרית) אוסף הדפים שניגשנו אליהם ב-*k* הגישות האחרונות לזכרון.

הערה אם הגישה היא אקראית, גודל ה-Working Set הוא בערך k . עם מקומות, גודל ה-Working Set קטן בהרבה מ- k . במקרה אפשר להתחיל מ-1 ולחגיל אותו עד שהגודל ה-Working Set מתיצב. במצבות יתיר הגיוני להסתכל על k מאוד גדול ולראות אילו דפים השתמשו בהם בקריאות הללו.

כדי לגלוות לאילו דפים ניגשנו, צריך תמייה מהחומרה. גישה לזכרון מתבצעת במהירות השעון ולכן צריך שהחומרה תעקוב אחריה. כדי שהתקורה תהיה קטנה יחסית, נשמר לכל דף שני בייטים. בית הרפרנס (האם ניגשנו לדף); ובית הלכלוך (האם שינו ערך בדף). ה-MMU אמון על ביצוע התיעוד הזה בכל גישה לזכרון. סכמא זו נתמכת ע"י מעבדי אינטל (וגם AMD).

אלג' גירוש נוספים

1. Not Recently Used - כל כמה זמן ננקה את ביתי הרפרנס של הדפים וכשהרייך לבחור קורבן, נבחר אקראית מבין אלו שיש להם בית רפרנס 0. ההיגיון הוא שהוא שדה לא שומש בערך בתקופה الأخيرة כנראה.

חסרו ההערכתה שלנו של הזמן האחרון שבו השתמשו בדף היא מאוד גסה.

2. Least Recently Used - נסדר את הדפים לפי הזמן האחרון שנענו בהם ונבחר בכל פעם את הקורבן שהשתמשו בו בעבר الأخيرة.

חסרו המימוש של LRU הוא מאוד יקר. אפשר לשמר Timestamp ווז מצוא מינימום בזמן לינארי. לחולפן אפשר להשתמש ב-Sorted List שזו סיבוכיות זיכרון גבוהה וכו'.

3. The Clock Algorithm (מה ששמשתמים בו ביום) - נדמיין שהמסגרות מסוימות בראשימה מעגלית עם מוחות שמכוון על אחד מהם. לכל דף יש בית רפרנס. כשרייך לארש דף, כל עוד הדף שעליו מצביעים יש בית רפרנס 1 נאפס אותו נמשיך הלאה ונאפס את הראשון שיש לו בית רפרנס 0.

האלג' הזה מזכיר את LRU, כי אם לדף יש בית רפרנס 0 זה אומר שעברנו סיבוב שלם (הרבה דפים נכנסו ויצאו) ועדיין לא השתמשנו בו, אז אפשר להיפטר ממנו.

4. Clock + Time Algorithm - נשמר את הדפים של כל תהליך בראשימה מעגלית עם זמן וירטואלי לכל אחד. בכל פעם שניגשים לדף נدلיק את בית הרפנס. ככל תיק של השעון, כל דף עם בית רפנס דлок מקבל במשתנה זמן וירטואלי שלו את הזמן הווירטואלי הנוכחי ומאפסים לו את בית הרפנס.

כשצריך לגרש מישחו, בוחרים את האחד עם הזמן הווירטואלי (זמן הגישה האחרון) הכى ישן.

אם בית הלכלוק כבוי, זה אומר שלא צריך להעתיק בחזרה לדיסק את הדף כי הוא לא השתנה. לכן,undyif לגרש דפים נקיים כי הם לא דורשים העתקה לדיסק, אלא רק מהדיסק את הדף החדש. לעומת באג' השעון, נדלג הלאה לא רק אם בית הרפנס דлок, אלא גם אם בית הלכלוק דлок.

יש מערכות הפעלה שעושות דפדף מקומי (כלומר מגרשות דף ששייך להתקלך הנוכחי) ויש כאלה שעושות את זה גלובלי (מגרשות את הדף הטוב ביותר מבין כל הדפים של מערכת הפעלה). היתרונו בדף גלובלי הוא שהוא פותר אותנו מלהשוו עלי כמה דפים להקצות כל תהליך.

תרגול

התרגול הוא חזרה אינסופית על מה שעשינו בהרצאה וזו שלפניה ולכון לא אעשה את כל הדוגמאות וההסברים שוב. בית אחד זה 8 ביט, KB זה 2^{10} בייטים, MB זה 2^{20} בייטים ו-GB זה 2^{30} – זה בנויגוד ליחידות בכלל שהן אלף- מיליון וכו', כאן הפעלים הם של (פי) 2^{10} .

$$\text{דוגמה} \quad 2^{20} \cdot 2^2 \cdot \frac{4\text{GB}}{8\text{KB}} = 2^{-1} \cdot \frac{2^{30}}{2^{10}}$$

דוגמה כמה מספרים ניתן לייצר במספר עם 8 בייטים? 2⁸ מספרים, כל הערכים בין 0 ל-1 – 2⁸ (בלי ייצוג המשלים ל-2).

כל מה שהמעבד עשו הוא או על הרגיסטרים או על הזיכרון. לכן נצטרך לנחל את הזיכרון בחוכמה כדי שהגישה של המעבד אליו תהיה יעילה כי הוא משענתית איתי יותר מאשר הרגיסטרים/מטמון).

הכתובת הפיזית היא המיקום של תא מידע ספציפי בזיכרון המרכזי (h-RAM). מרחב הכתובות הפיזיות האפשרי תלוי בגודל הזיכרון.

$$\text{דוגמה} \quad \text{זיכרון עם גודל } 8\text{GB} \text{ דורש כתובות בגודל } \log_2 8\text{GB} = .33$$

חלוקת של הזיכרון לאזוריים לכל תהליך וכל אחד ישמש כמו שהוא רוצה. הבעיה עם שיטה זו היא שאנו צריכים לדעת אילו תהליכי מגיעים ומתי, שזה בסדר במערכות כמו מקרים חכמים כי הכל ידוע מראש, אבל זה לא עובד במחשבים אישיים.

את הכתובות של כל תהליך נרצה לקבוע בזמן ריצה, לפחות-central יותר, לפעמים פחות, ולכון התהיליך לא יודע מה הכתובות הפיזיות שלו בזמן קומפליציה. כדי שהרעיון הזה יוכל לעבוד, מספר לכל תהליך שיש לו זיכרון (ווירטואלי) רציף לכל שיירת ומאהורי הקלעים מסדר את העניינים בעצמו (במערכת הפעלה). עכשו יצרנו לעצמנו בעיה חדשה - איך נתרגס כתובות וירטואליות פיזיות?

מי שעושה את הקורדיינציה הזו בין התוכנה לחומרה הם h-MMU בעוזרת מערכת הפעלה.

מרחיב הכתובות הווירטואלי מוגבל בארכיטקטורת מערכת הפעלה (32-ביט מאפשר 4 ג'יגה ו-64-ביט הרבה יותר).

סגןנטציה

הרעיוון בסגמנטציה הוא שתוכנה היא בסה"כ אוסף סגמנטים (קוד, משתנים, מעתנים סטטיים, פונקציות ספריה וכו') וכן אפשר להקצות לכל תחליק זיכרון לפי סגמנטים. במקרה זה, כתובת לוגית היא טאפל מהצורה (seg_num, offset) ואת התרגומים מבצעים עם טבלת סגמנטים, כאשר בכל כניסה של הטבלה יש לנו את כתובת הבסיס והחסם העליון של הסגמנט. נשמר בנוסך לכל כניסה בית ולידציה (אם הסגמנט טוען לזכרון ולא יושב בדיסק) וגם כל מיני מידע על פריוויליגיות כתיבת/הרצה וכו'.

הערה ה-MMU יודע מה טבלת הסגמנטים של התהליק הנוכחי באמצעות ה-Segment-Table Base Register ואות מספר הכניסות בטבלה באמצעות ה-STLR (Base Length). במקרה לזכרון, נדרש שמספר הסגמנט יהיה קטן מערך ה-STLR וגם שההיסטוריה יהיה קטן מהחסם העליון של הסגמנט.

הבעיה המרכזית עם סגמנטציה היא פרוגרנמנטציה חיצונית, ככלומר שנוצרים חורים קטנים בלתי ניתנים להקצאה בין הקצאות לסגמנטים.

דפדוֹן

הפתרון לבעה זו הוא דפדוֹן. כל דף (בזיכרונו הווירטואלי) הוא בגודל קבוע ומתאים למסגרת (בזיכרונו הפיזי). בכלל שהגודל קבוע אין פרוגרנמנטציה חיצונית.

כתובת לוגית היא טאפל (d, p) כאשר p הוא מספר הדף ו- d הוא ההיסט בדף הדף. מספר הביטים להיסט הוא \log_2 של גודל הדף ובהתאם גם מספר הביטים למספר הדף.

גם בטבלאות דפים נשמר מידע נוסף - בית ולידציה, האם לדף יש מסגרת; בית שינוי - האם שונה ערך בתוך הדף; בית רפרנס - האם ניגשנו לערך בתוך הדף ועוד.

כל תהליק יש טבלת דפים מסוימת וכלן התקורה (בזיכרונו) יכולה להיות כל כך גדולה שאין מקום לאחסן את כל בטבלאות הדפים בזיכרונו.

דוגמא אם כל כניסה בטבלת דפים היא 4 ביטים (סביר מאד), אז במערכת של 64 ביט יש מקום ל- $\frac{2^{64}}{2^{32}}$ דפים, ככלומר 2^{52} . לכן נדרש 2^{52} כבאים בטבלאות, שזה 2^{54} כתובות זיכרון, שזה כמונן הרבה מעבר למה שיש לנו.

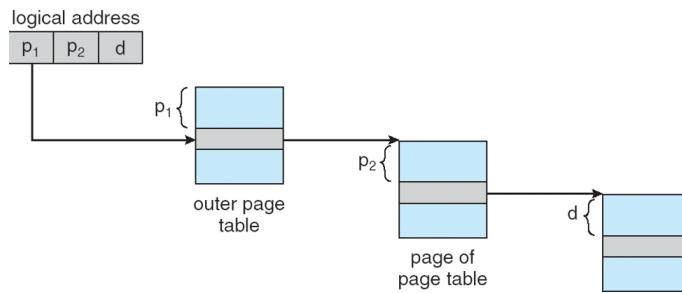
הערה המצב הזה ריאליסטי כי יכולם להיות אלפי תהליקים שרצים במחשב בו זמן-

טבלת דפים היררכית

במקומות שתהיה לנו טבלת דפים אחת גדולה, נשמר ערך של בטבלאות דפים כאשר כל טבלה מצביעה לטבלה נוספת. ככלומר טבלת הדפים מדופדפת. החסרון כאן הוא שנוספו לנו עוד קרייאות לזכרון (בעומק 2, יש שלוש קרייאות כבר). במימושים של טבלת דפים היררכית משתמשים גם ב-TLB באופן האופן כמו במקרה הסטנדרטי.

דוגמא נסתכל על מערכת 32 ביט עם 1KB זיכרון. גודל ההיסט הוא 10 ביטים וגודלו מספר הדף הוא 22 ביטים. בכלל שככל טבלת דפים היא בעצמה מדופדפת (כלומר כל תת-טבלה יושבת בדף בודד) נחלק את מספר הדף ל-10 ביטים למספר הדף ברמה השנייה ועוד 12 ביטים להיסט בטבלת הדפים השינוינית.

לכן בהינתן כתובת וירטואלית (p_1, p_2, d) , נמצא את הערך באינדקס p_1 בטבלה החיצונית. הערך הזה הוא האינדקס של הדף (של טבלת הדפים) שאנו צריכים. בתוך הדף (בעומק 2), נמצא את הערך באינדקס p_2 . זה הדף האמיתי שלנו, ובתוכו ניגש לערך ה- d במסגרת המתאימה לדף.

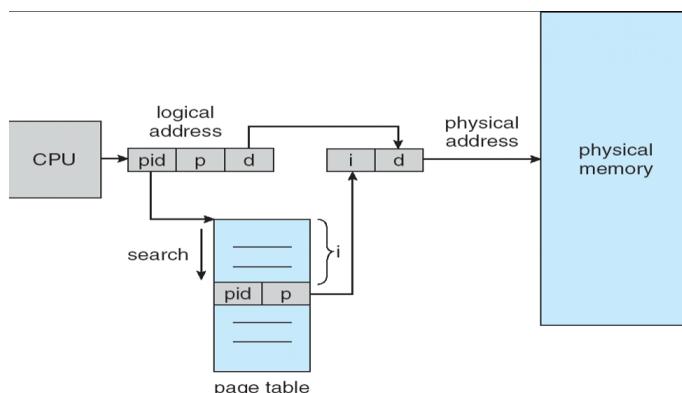


איור 62 : גישה בטבלת דפים היררכית בעומק 2

הערה מערכת הפעלה אמונה על ייצור העץ/טבלה זהו, אבל ה-MMU הוא זה שמשתמש בה בפועל.

טבלת דפים הפוכה

נשמר טבלת דפים אחת לכל התהליכים יחד. הטבלה שומרת עבור כל מסגרת (בזיכרון הפיסי) את הדף שתואם לה, לרבות לאיוזה תחילה שיעיד הדף. זה מקטין את תקורת הזיכרון אבל מבচינת תרגום, הסיבוכיות עולטה בהרבה (צריך לעבור על כל המסגרות). עתה הכתובת הלוגית תכיל גם מספר דף וגם מספר תחילה, כאשר בהינתן כתובות לוגית נבדוק האם הדף נמצא בטבלת הדפים החופча, אם לא נזרוק PAGE FAULT, אבל אם כן אינדקס הכניסה שבה מצאנו את הדף הוא אינדקס המסגרת הנדרשת, ומשם מחשבים את הכתובת הפיסית כרגע.



איור 63 : גישה לטבלת דפים הפוכה

שבוע XX | דפודן לעומק

הרצאה

אנחנו מדברים עדיין על דפודן (על בסיס דרישת) שבו מתרגמים כתובות, אם הדף לא ממופה זורקים PAGE FAULT, מביאים את הדף מהdisk וקוראים חזרה לתהילך.

מה התקורה של דפודן? נסמן ב- p את ההסת' ל-PAGE FAULT, לכן זמן התגובה האפקטיבי הוא

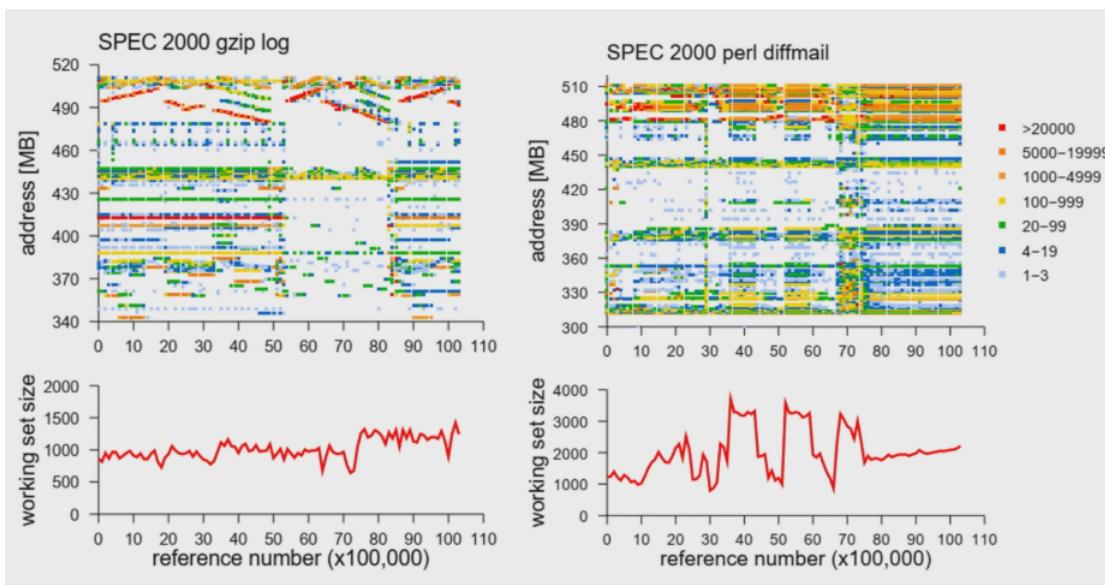
$$p(\text{FAULT PAGE}) + (1 - p)(\text{זמן גישה ל זיכרון})$$

וההאטה היא זמן גישה האפקטיבי חלקי זמן גישה ל זיכרון.

לרוב לוקח 25 מיל-שניות כדי לטפל ב-PAGE FAULT וכ-100 נאנו-שניות כדי לגשת ל זיכרון. אם $p = 0.001$ ההאטה היא פי 250 ולכן ככל שחשוב לנו אלג' גירוש איקוטי. אם $p = 0.0000004$ אז ההאטה היא 1.1 שזה יחסית טיפוסי.

הערה הביצועים תלויים בראש ובראונה ב מהירות דיסק.

דוגמה באירר רואים את הביצועים של תוכנות שמוסכמת שהן מייצגות ביצועים טיפוסיים (benchmark). כל שהצבע יותר חם (למעלה) זה אומר שימושים יותר בכטובת. ציר ה- x הוא ציר הזמן (אינדקס הגישה) וציר ה- y הוא מקום הכתובת.



איור 64 : ניתוח ה- W set בתוכנות working set benchmark

ניתן לראות כי גישה ברוב המיקומות היא מאוד מוקדמת, כאשר האלכסונים הם גישות למערכים. למטה רואים את גודל ה- W set, working set, ונitinן לראות שהוא משתנה אבל עדין יחסית יציב, תלוי בתוכנה. כל זה מוכיח את עיקרונו הלוקאליות, מבחינת המקום בחלק של המערכים ומבחן הזמן כשניגשים כל הזמן ל זיכרון האדום (הקו האחד).

локאליות בזמן

локאליות בזמן מייצגת שני דברים שונים :

- Clusters accesses - כשניגשים למשתנה אחד הרבה, ואז משתנה אחר הרבה וכו' וכו', כאשר אין חזרה על משתנים. זה מה שנאיבית היינו חשובים שיקרת.

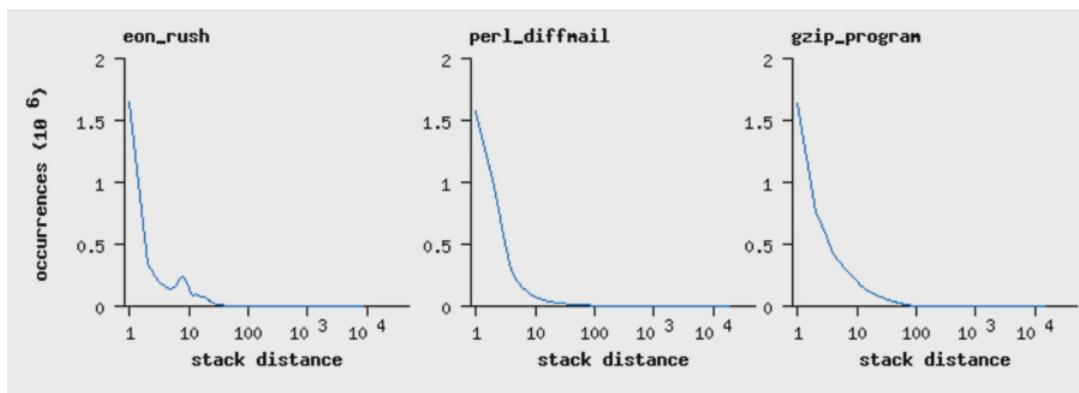
- פופולריות מוטה - כשניגשים לששתנה אחד (או יותר) הרבה באופן לא פרופורציוני לאחרים במהלך הרצאה שבו בין הגישות אלו נינגים גם לדברים אחרים. זה קורה לדוגמה עם אינדקס בלולאה שמתעדכן כל הזמן ואילו משתנים אחרים רק לעיתים/בכלל לא.

פופולריות היא תופעה שאנחנו רואים הרבה מאוד במדמ"ח, ככלומר בשיטתיסטייקה לגישה למשאבים מוטה מאוד. כאן מדובר בששתנים פופולריים אבל כך גם בנוגע לקבצים, אתרים שניגשים אליהם הרבה לעומת אחרים נישתיים. הדבר הזה נגרם מהקצתה של הפערים, העשירים מתוערים והענינים מתענים.

הערה כדי לתאר את ההטיה הזה הומצאה התפלגות *zip*. הפופולריות של המילה ה-*i*- פופולריות ביותר בשפה האנגלית היא $c(i) \propto \frac{1}{i}$.

כיצד ניכמת לוקאליות? נזכיר לאן ניתן להתחילה לאורך הרצאה שלו. נשים את כל הכתובות האלה במחסנית. בכל פעם שהתחילה יגש לזכרוון, נבדוק האם הכתובת נמצאת במחסנית ונשים אותה בראש המחסנית. כך יוצא עמוקה הכתובת במחסנית מעיד על הרלוונטיות הזמנית של הכתובת, ככלומר האם היא חלק מהлокאליות הזמנית (אם יש כזו). אם אנחנו ניגשים באקראיות לכתובות נצפה שנצטרך להיכנס די עמוק למחסנית.

דוגמה באירור הבא ניתן לראות את עומק הכתובת במחסנית בכל גישה לזכרוון של אותן תכניות benchmark. אכן ניתן לראות כי כמעט הכתובות ניגשות יחסית גבוהה במחסנית.



איור 65 : ניתוח הלוקאליות הזמנית בתוכנות benchmark

הניסוי המחשבתי של עומק המחסנית הוא בעצם דיא מייעשי - LRU ניתן למימוש באמצעות החזקת מחסנית שעובדת כפי שתיארנו (על דפים במקומות כתובות מדויקות) בגודל k (מספר הדפים) ולגרש כל מה עמוק יותר מאשר $m-k$. במקרה כזה, ההסת' ל-T-LFAULT היא החסת' שיש יותר מאשר $m-k$ איברים במחסנית.

לא תמיד הגישה היא רק לששתנים ייחדים, אלא לעיתים למבני"ת שדוגמים ל-stream-stream, לדוגמה באפרים מאוד ארוכים. יש לנו לוקאליות מרחב (ניגשים לכתובות קרובות, צמודות למשהו) אבל אין לוקאליות בזמן כי לא קוראים את המערכת שוב (בטח לא בנסיבות שעליהם אנחנו מדברים). באפרים כאלה דורשים הרבה דפים ויגרמו לגירוש הרבה מאוד דפים בלי שום סיבה טובה.

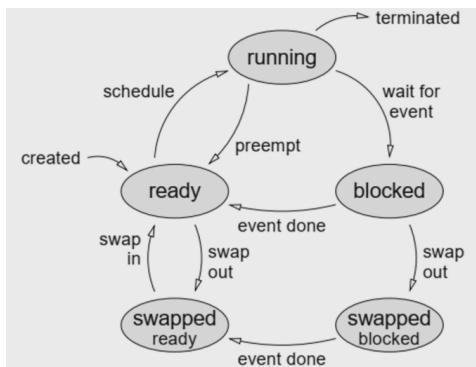
לכן היינו רוצים לויהו איזה דף הוא פעיל ואיזה לא, ואם הוא לא פעיל לזרוק אותו מיד לאחר השימוש. אלג' כזה אכן ממומש במערכת הפעלה לינוקס (הרחבת לאלג' השעון), בו דף שמסומן בתור לא פעיל מקבל הזדמנות נוספת ואם הוא פעיל לא מגרש אותו, ואם אכן מתברר שהוא לא כן זורקים אותו.

Swapping ו-Thrashing

כל שיש לנו יותר multiprogramming כך הניתלת של המעבד עולה אבל עד דרגה מסוימת. החל מנקודת מסוימת הביצועים יורדים דרסטית כי יש כל כך הרבה תהליכיים שככל אחד צריך דף אחד והוא מגרש דף של אחר וуд שנחזר לתהליך הקודם כבר הדפים שהוא צריך יהיו לו PAGE FAULT וחוזר חלילה. תופעה זו נקראת **thrashing**.

כדי למנוע תופעה כזו, נבחר תהליכיים מסוימים ונגרש את כל הדפים שלו לפרק זמן כלשהו כדי שאחרים יוכל להשתמש בו, ואז נחזיר אותם בשתקופת הייבש שלו והוחלט שתיגמר.

בחינת הניתוח של מערכת כזו, נוסיף עוד מצבים לדיאגרמת המצבים שראינו בעבר (באיור המעודכנת).



איור 66 : דיאגרמת המצבים המעודכנת

כאן אנחנו עושים swap (גירוש כל הדפים) לתהליכי רק אם הוא חסום, וזאת עד שהairoう שהוא מחכה לו קורה, ואז הוא מוכן שייעשו לו swap פנימה ולחזור לרוץ. המעבר בין השורה התחתונה לאמצעית ולהפוך מבוצע באמצעות המתזמן לטוחה הבינוני, המעבר בין השורה הראשונה לשניה ולהפוך מבוצע באמצעות המתזמן לטוחה הקצר. בנוסף, נשים לב כי המתזמן לטוחה הקצר מנהל את הגישה למעבד בעוד המתזמן לטוחה הבינוני מנהל את הגישה לזכרון.

במערכת 32 ביט, אם כל דף הוא 4KB אז לכל תהליכי יש 2^{20} דפים, שזה אומר שכל כניסה בטבלת הדפים היא 4 בתים שזה אומר שטבלה הדפים צריכה 1000 דפים, שעצם תופסים מקום בזכרון כמו כל דף אחר.

במערכת 64 ביט הטבלה דורשת 35 מגה ביטים שזה די חמור. גם במעבדים שבהם משתמשים אפקטיבית ב-48 ביט עם דף בגודל 4KB הטבלה לוקחת עדין כמה מגה בייטים. אם הזיכרון הפיזי הוא 32 ג'יגה, כלומר 2^{35} דפים ולכון לפחות $2^{23} - 2^{43}$ מה-entryים הם תמיד לא חוקיים (מיילוניים מכל הטבלה). לכן נדרש להשתמש בשיטה יותר טובה.

טבלת דפים היררכית

במקום שתיהיה לנו טבלה אחת שטוחה, נשמר טבלה שהיא שורש והיא כולל מידע על דפים שמיכילים בטבלת דפים נוספת וכך רקורסיבית, זה מה שאנו ממשים בתרגיל 4. כך לא צריך להקצות כל הזמן את כל הטבלה אלא רק את הטבלאות שימושיות אותן ניתן לגישה לדפים של התחלת.

דוגמא כדי לגש כתובות (p) בטבלה היררכית בעומק 2, ראשית ניגש לפריים שמוופיע בכניסה ה- p_1 -ית בשורש ומשם לפריים שמוופיע בכניסה ה- p_2 -ית ובפריים הזה ניגש לכתובת $offset$ וכן תרגמנו בהצלחה את הכתובת הווירטואלית לפיזית.

דוגמה נתרגם את הכתובת $(5200, d)$ כאשר 5200 הוא חלק של 20 בתים ו- d הוא 12 בתים. נניח שיש לנו טבלה היררכית בעומק 2 . לכן כל אינדקס הוא 10 בתים, עם חישוב פשוט מקבלים שזהו $(5, 80, d)$.

נבצע את התהליך בדוגמה הקודמת ונקבל את הכתובת הרצויה.

הבעיה כאן היא שהוספנו הרבה גישות לזכרון, שזו תקורה גבוהה (לעומת קריאה אחת קלאסית בטבלה שטוחה). בנוסף, אם הדפים של טבלת הדפים לא בזיכרונו הם גם זורקים PAGE FAULT ונכנסים לוורטקס אינסופי. יתרה מכך, רוב הטבלאות בעומק 2 ומעלה הן invalid כי לא השתמשנו בהן עד עכשו.

טבלת דפים hashed

כדי לפטור את הבעיה הנ"ל, במקרים להשתמש בטבלה היררכית/קלאסית, נשתמש בטבלת האש כאשר המפתחות הם מספר הדף, באמצעות מייעון פתוח (Linked list) בכל תא). כך נמצמנו את כל האפשרויות לכטובות לטבלה בגודל קטן יותר והגישה הרבה יותר מהירה (רק פעם אחת).

טבלת דפים הפוכה

אם מרחב הכתובות הווירטואלי גדול בהרבה מהגודל הפיזי, נוכל במקרה לשחק את הזיכרון הווירטואלי ולזכור איזה דף ממופה לאיוז פריים, נבצע העתקה הפוכה.

לכל פריים נזכיר את הדף שהוא שומר כרגע יחד עם ה-*pid* של התהיליך שלו שייך הדף הזה, שכן שני דפים יכולים להיות אותו אינדקס דף 0 (לדוגמה, יש לcoldם) אבל עם תכנים שונים לגמרי).

כאשנחנו מקבלים כתובת (pid, p, d) לתרגום, ה-MMU עובר על כל טבלת הדפים בחיפוש לינארי, אם הוא מוצא את הדף מעולה ואחרת זורק PAGE FAULT ומהם כבר אנחנו מכירים.

הגנה ושיתוף

כל דבר שמצויה בטבלת הדפים נגיש לתהיליך אבל כל מה שלא, לא קיים לדעת המעבד. לכן, המעבד לא יכול לגשת לכטובות של תהיליך אחד כשהוא מרץ תהיליך אחר. המגבלה הזה היא פיזית - זה טכנית לא אפשרי. לעומת זאת, שינוי של טבלה הזיכרון משנה את איך שאנחנו משתמשים על הזיכרון (פעולה זו היא כזכור חלק מהكونטיקסט סוייז').

אותו הרעיון עובד גם לכיוון השני. אם יש לנו טבלת דפים היררכית, נניח שני תהילכים רוצים שיהיה להם זיכרון משותף, אז מספיק שהטבלה בעומק 1 (השורש) של שני התהילכים תקבע על אותה הטבלה השינויית ותהיה להם גישה משותפת לדפים בתוך הטבלה הזו, אבל עדין הפרדה בכל תוכן אחר של הזיכרון.

נסכם את כל מה שעשינו בנושא הדפודוף וניהול הזיכרון,

פתרון	בעיה
התהיליך רואה (זיכרון) שונה מאוד מהמציאות	תרגום כתובות באמצעות סגמנטציה/טבלאות דפים
איחוד סגמנטדים/דפודוף	פרגמנטציה חיונית
זיכרון הלוגי גדול בהרבה מהזיכרון הפיזי swapping	

טבלה 4 : סיכום של דפודוף

קובציים

קובץ, בингוד לכל מה שראינו עד כה, הוא אבסטרקציה (לא היה קיים בכלל בלי ה-OS) בינגוד למה שראינו לאחרונה (ניהול זיכרון, ניהול מעבד) שזה היה וירטואלייזציה. קובציים הם אחת האבסטרקציות המרכזיות של מערכת הפעלה, והם מספקים ממשק נוח למשתמשים ולאפליקציות.

ראשית ננסה להבין מה המהות של קובץ. קובץ, בינגוד לזכרון ורגיסטרים, נשאר תמיד, גם אם מכבים את המחשב. פורמלית, קובץ הוא אחסון מידע **משמעותי** (יש משמעות לתוכן ולסדרו), **פרטי** (נשאר גם אם מכבים את המחשב) ובעל **שם** (ניתן לזהות אותו באמצעות שם). שמות משמשים אותנו גם בקומפקטים אחרים במחשב (בגישה לאטרים) ואפילו בגישה לזכרון מסוות של תהליכיים (הם צריכים לדעת איך לזהות לSEGMENT המשותף).

הפריסיטנס מתבטאת בשמירה של הקובציים על התקנים שהם לא volatile (דיברנו על זה בהקשר של מטמון), לדוגמה HDD או SSD והמידע נשמר גם כשהמחשב כבוי. מעבר לכך, חשוב שהמידע ישרם גם אחרי שהתהליך שייצר אותו נגמר (כי אחרת מה עשו בו זה).

הערה אמם שימור לארוך זמן נשמע כוללני, אבל זמן החיים של מדיה דיגיטלית הוא לרוב כמה שנים. מעבר לכך, לעיתים אין מכשירים שאפשר להשתמש בהם כדי לקרוא את המדיה וגם אם כן, אנחנו לא תמיד יודעים/זוכרים איך לפרש את הביטים (זה פורמט עתיק לסרטון של חתול? פורמט עתיק לתמונה של חתול?).

ביוניקס קובץ הוא סדרת בתים בלי שום משמעות נוספת, וזה המטלה של האפליקציה שימושה בקובץ לפריש/לפענה אותו. במחשבים של IBM, קובץ הוא רצף של רשומות והוא אפשר לגשת לפי רשומות באמצעות מערכת הפעלה ולא רק לפי בתים. בווינדוס, מערכת ה-NTFS מתייחס לקובץ כאוסף מפתחות-ערכיהם (לדוגמה האיקון של הקובץ) וגם סטרים של מידע בלי שם (תוכן הקובץ לרוב).

נדמה לנו שימושים שיש קשר בין extension של הקובץ לתוכנו, אבל זה לא בהכרח המצב. ביוניקס אפשר לשים סיומות אבל הוא יתעלם מהן. בווינדוס יש לזה משמעות כי מערכת הפעלה מנסה לשיקן בין אפליקציות שתומכות בסיומת לקובציים עם הסיומת. במקרה הכללי, מערכת הפעלה נותנת תשתיית לנתונים עם קצר מטא-דאטה אבל בלי להגביל את האפליקציה.

האפליקציה מפרקת ומעבדת, יוצרת ומשתמשת וקובעת מה לעשות עם הפורמט/התוכן.

מבחינת מערכת הפעלה, מערכת הקובציים מתחלקת ל-API שוחף לאפליקציות לביצוע הפעולות האמורויות, והמיומש הפנימי של המערכת הקובציים.

תרגול

אנחנו עוסקים באlg' גירוש. כשל הדפים תפוסים וצריך להכניס אחד חדש, צריך לגרש דף ולהכניס אחר מהdisk. כל PAGE FAULT לוקח הרבה זמן לטיפול ולכן לזרע את תדירות פיסוף הדפים (page miss rate).

הערה נדונן באlg' גירוש שונים, בהיעדרם מהשימוש, כי העיסוק עם גירוש דף של הטבלה היררכית הוא מיותר (זה מה שעושים בתרגיל).(4)

שיטות גירוש

- האופטימלי: נבחר את הדף שלא השתמש בו הכי הרבה זמן. אי אפשר יותר טוב מזה אבל הבעיה היא שאנו לא יכולים למש את זה כי אנחנו לא יודעים את העתיד. השתמש בו להשוואה.

דוגמה נניח שאנו ניגשים לדפים בסדר הבא משמאלי לימין 0, 0, 1, 2, 1, 6, 4, 0, 1, 0, 3, 1, 2, 1. עד הרביעי הכל בסדר. בהכנסת 4, נכניס את 6 כי אנחנו לא הולכים להשתמש בו יותר. 0 ו-1 כבר בפנים ולכן לא צריך לעשות כלום. לאחר מכן 3 צריך להיכנס במקום 4 או 0 (לא משתמשים בשנייהם) וכו'.

$$\text{תדירות ה-TFAULT} = \frac{6}{12} = 0.5$$

- FIFO: נוציא את הותיק ביותר בזיכרון.

דוגמה על אותו הסדר מלמעלה, נוציא את 0 בשביל 4, אז את 2 בשביל 0 וכו' כך שסה"כ תדירות השגיאה היא $\frac{9}{12} = 0.75$.

הבעיה כאן היא שהעובדת שימושו יושב בזיכרון הרבה זמן לא אומר לנו לא הולכים להשתמש בו בהמשך, להפוך.

- Second Chance FIFO: למדנו אותו כאlg' השעון. נשמר לכל דף בית רפרנס R שנדייל כל פעם שניגש אל הדף. נרים רק שאם על דף מועמד לגירוש, נapse $R = 0$ ונחזיר אותו לסוף התור (אילו הוא חדש).

- LRU: נרש את האחד שלא השתמשנו בו הכי הרבה זמן. זה מושכל יותר מ-FIFO כי מה שלא השתמשנו הרבה זמן בו לרוב לא השתמש בו בעבר. בהינתן המידע שיש בכל איטרציה, זו הבחירה הכי מושכלת.

דוגמה כדי להכנס את 4 נוציא את 0, כדי להכנס את 0 שוב נוציא את 2, כדי להכנס את 3 נוציא את 6 וכך להכנס את 2 נוציא את 4. תדירות השגיאה הפעם היא $\frac{8}{12} = 0.67$.

הבעיה עם LRU היא שלזכור מה הדף האחרון שניגשנו אליו זה מאוד קשה. אפשר להשתמש בקונטרים לכל דף, או לשמור הכל במחסנית, אבל זה קשה בחומרה ולכן הרבה משתמשים בשיטות שמקרובות את LRU.

שאלה נניח שיש לנו מכונה עם 8GB (2^{33}) זיכרון פיזי וגודל דף של 8KB (2^13), עם כניסה בגודל 4 ביטים (2^2) בטלת הדפים.

- כמה רמות בטלת דפים היררכית נדרשת כדי להתאים כתובות וירטואלית באורך 46 ביט אם כל טבלה נכנסת לדף יחיד (וכל מילה היא בית אחד)?

היחסט בכל דף הוא בגודל 13 ביטים. בטללה אחת יש לנו $2^{13} = \frac{2^{13}}{2^2} = 2^{11}$ כניסה. לכן התשובה היא 3 רמות, כי הכתובת היא $(11, 11, 11, 13)$ כאשר הגדים הם ($p_1, p_2, p_3, offset$).

- פרטו את השדות של כל כניסה בטללה.
- בכניסה יש 32 ביטים. מספר הפריים בו נמצא הדף של הכניסה מוצג ב-20 ביטים (גודל הזיכרון חלקי גודל פריים היחיד).

- בלי מטמון, כמה פעולות זיכרון נדרשות כדי לקרוא/לכטוב למילה אחת באורך 32 ביטים?
- 4, כדי לצורך את הפריים בכל אחת מהטבלאות (3) וקריאה בתוך הפריים עצמו להיחסט (1).
- כמה זיכרון פיזי נדרש לתהילך עם שלושה דפים בזיכרון הווירטואלי?

לשימוש 48KB לשישה דפים, שלושה לטללה (שורש, עומק 1, עומק 2) והרמה الأخيرة תקבע לשלוות דפי ה"תוקן".

שאלה ענו על השאלה הבאה.

התבונן בקטע קוד הבא שמאפס מערך של מספרים מסוג `integer` (כל `integer` הינו בגודל 4 בתים).

```
for (int i=0; i<2^29; i++) {  
    numbers[i] = 0;  
}
```

הנחות:

- למחשב זיכרון פיזי בגודל 2^{32} בתים המוחלך למסגרות בגודל 2^{12} בתים. שימוש לב:
כל מסגרת מכילה עד 2^{10} איברים מסוג `integer`.
- איברי המערך מוקצים בצורה רציפה בתוך כל דף ומתחילה מתחילה הדף הראשון.
- הקוד כולל כניסה לדף אחד.
- שיטת החלפת דפים הינה `paging demand` - ? נמצא ברגיסטר.
- בהתחלה הזיכרון מכיל רק את טבלאות הדפים והן נשארות תמיד בזכרון (התעלמו מ-`page faults` על טבלאות הדפים).

א. (6 נק') נניח שהחלפת דפים מתבצעת במדיניות LRU. כמה `page faults` יהיו במהלך האלגוריתם אם מוקצים לתהיל' 2^{12} מסגרות בזכרון (לא כולל טבלאות דפים)? נמק.

איור 67: סעיף א' בשאלת ממבחן

בכל פעם שצריך דף חדש, LRU יוציא את הדף הכי מוקדם ששחטמשנו בו. המערך כולל מתרפרס על 2^{19} דפים ולכון כל התהילה נמצאת על $1 + 2^{19}$ דפים. הדפים לא בזכרון בהתחלה ולכון צריך להביא את כלם לזכרון בהתחלה, ככלומר יש PAGE FAULTים במספר הדפים, כאמור $1 + 2^{19}$.

ב. (6 נק') נניח כעת שהחלפת דפים מתבצעת במדיניות Second Chance FIFO. כמה `page faults` יהיו במהלך האלגוריתם אם מוקצים לתהיל' 2^{12} מסגרות בזכרון (לא כולל טבלאות דפים)? נמק.

איור 68: סעיף ב' בשאלת ממבחן

ראשית כל עוד יש דף פניו,ऋיך להביא את כל הדפים (כולל הקוד) מהזיכרון כלומר 2^{12} הקצאות עד האיתרציה- $(2^{12} - 1) \cdot 2^{10}$ של הלולאה (2^{10} מספרים בכל דף, $1 - 2^{12}$ מסגרות למערך בלי הקוד). בגלל האופן שבו אנחנו ניגשים לזכרון, ככלם תמיד מקבלים הזדמנויות שנייה ולאחר מכן פשוט ב-FIFO, ככלומר כל 2^{12} PAGE FAULTים, נגרש את דף הקוד שמידऋיך אחריו. ככלומר יש לנו 2^{19} PAGE FAULTים נספחים לפעם הקודמת. לכן התשובה היא $2^7 + 1 + 2^{19}$.

שאלה נתונה סדרת הגישות לדפים לפי הסדר הבא (משמאל לימין, אינדקסים למטה לנוחות)

1	2	3	4	1	3	2	1	5	2	3	6	5	3	2	1
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

טבלה 5 : סדר הגישות לדפים בשאלת

מלאו את הטבלה הבאה בהנחה של זיכרון הפיזי יש 4 מסגרות.

	קורבן 1	קורבן 2	קורבן 3
LRU	4	1	6
FIFO	1	2	3
Second Chance FIFO	1	4	5

טבלה 6 : קורבות לפי אלג' גירוש שונים

ב-LRU נגרש את 4 כי לא משתמשים בו יותר שמכניס את 1 באינדקס 9, נגרש את 1 כי הוא האחרון שאנו משתמשים בו מבין הדפים שכרגע בזיכרון. בסוף נוציא את 6 כי לא משתמשים בו יותר.

ב-FIFO ברור שמנורשים לפי הסדר השלושה הראשונים.

ב-FIFO עם הזדמנויות שנייה, ראשית נגרש את 1.

כינאפס לדפים $4 - 1$ את R ואז נגיע ל-1 שוב. לאחר מכן נרצה להכניס את 6 אז נאפס את R של כל הדפים חוץ מ-4 כי בו לא השתמשנו מאז הגירוש האחרון ולכן הוא זה שיגורש. בסוף באינדקס 16 כ שנרצה להחזיר את 1, נזרוק את 5 כי הוא כموון האחרון ששומש וזה- R של כולם היה דлок.

שאלה התבוננו בקטע הקוד הבא במערכת 32 ביט. הפ' יוצרת תהליך חדש וערק ההחזרה שלו הוא 0 אם אנחנו כרגע בתהליכי החදש ואחרת ה-pid של הבן. התהליכי החදש ממשיק את ריצת הקוד כמו ההורה שלו. קלומר הבן הוא זה שייכנס ל-if וההורה יכנס ל-else

```

int main(void)
{
    int var = 0;
    int i = 0;

    printf("%d %p\n", var, &var);

    for (i=0;i<3;i++)
    {
        if(fork()==0)
        {
            var = 2*var;
            if (i==2) { printf("%d %p\n", var, &var); }

        }
        else
        {
            var = 2*var + 1;
            if (i==2) { printf("%d %p\n", var, &var); }
        }
    }

    return 0;
}

```

איור 69 : קוד משאלת ממבחן

הקוד יוצר כמו וכמה תהליכיים (כל בן יוצר גם בניהם אחרים) ומדפיס רק באיטרציה الأخيرة את *var* וכתוובתו בכל אחד מהתהליכיים (עם race condition אפשרי).

תזכורת: **ק% בפקודת printf מדפיס מצביע בבסיס הקסדצימלי ומוסיף אפסים משמאלי** לפי הצורך. כל ספרה בסיס זה מייצגת 4 ביטים (בינהרמים) - למשל הספרה 7 מייצגת את הביטים 1110 והספרה E את הביטים 1110.

נניח כי פקודות printf ו-fork אין נcessות והשורה הראשונה שהודפסה בפלט היא:

0 EA655A78

א. (8 נק') כתבו פלט אפשרי (השורות שיודפסו ע"י פקודת printf) בסיום הריצה. (**אין** **לכתוב שוב את השורה הראשונה**)

איור 70 : סעיף א' משאלת ממבחן

נשים לב כי הכתובת של *var* היא זהה לכל תהליך כי *fork* מעתיק את מרחב הכתובות הווירטואליים של התהליך החורף, וזה לא משתנה רק הפיזי אבל זה לא מעניין, הכתובת המוצגת היא וירטואלית).

נציג אפשרות אחת להדפסה. מתחילה *i* = 0, *var* = 0, *i* = 1, *var* = 0-1, *i* = 2, *var* = 0-2, *i* = 3, *var* = 0-3 וכו' וכך יוצר נזן שגם מקיים 0-0 *i* = 0, *var* = 0-0 (והכתובת הקבועה).

נחזיר אחורה לנכד, הוא מעדקן $var = 1$ ומדפיס 1. באירועזה הבאה שלו הוא יוצר עוד בן שמדפיס 2 וכו' כך ששה'ב נקלט הדפסה של הספרות מ-0 עד 7. בעץ כולם יש 8 תהליכיים.

- כמה פלטיטים אפשריים יש לתהליך?

!8 כי כל סדר של התהליכיים יכול להופיע.

- מה גודל מרחב הכתובות הווירטוואליות (במילים) המוקצה לכל תהליך? אם לא ניתן לדעת, הסבירו מדוע.

הכתובת של var ניתנה לנו ב-32 ביטים, כלומר מרחב הכתובות הוא 2^{32} .

- מה הגודל של הזיכרון הפיזי במילימטרים במערכת? אם לא ניתן לדעת, הסבירו מדוע.
אין דרך לדעת, לא התקבלו נתונים עליו.

שבוע II | מערכות קבצים

הרצאה

האפליקציה קבץ מתחמשת עם ה-OS באמצעות syscall-s. כך גם מערכת הקבצים עובדת, יש לפתח קובץ, סגירה, קריאה, שינוי הרשאות ואטיריבוטים, יצירה קובץ, seek (הזוזת הפוינטර בקובץ) וכיוצא-ב.

הערה אין העתקה חד-ערכית ממערכת הפעלה למערכת קבצים וכל מערכת הפעלה יכולה (וakan לרוב זה המצב) לתמוך במערכות קבצים שונות. כדי שנוכל לחבר דיסק-און-קי לכל מחשב שהוא, נוצר שמערכות הפעלה של המחשבים האלה יכירו ויידעו להשתמש במערכות הפעלה שמותקנת על הսטיק.

לכל קובץ יש מטא- данא שמנוהל על ידי ה-OS. ביןיהם שם, גודל, הרשאות, פוינטר למיקום בדיסק וכו'.

שמות במערכת קבצים

לכל קובץ יש שם, והוא נמצא מבנה נתונים היררכי שמכיל תיקיות ותתי-תיקיות (directory בלינוקס/יוניקס ו- folder לוינידוס). זה לא תמיד היה המצב (מערכות קבצים ישנות לא היו בנויות כהה).

חלוקת ההיררכיה יש כמה מטרות:

- יעילות: איתור קובץ במהירות בלי צורך לעבור על רשימה עם כל הקבצים.
- הפרדה: שני משתמשים יכולים לקרוא לקבצים שונים באותו שם ושמות שונים לאותו הקובץ.

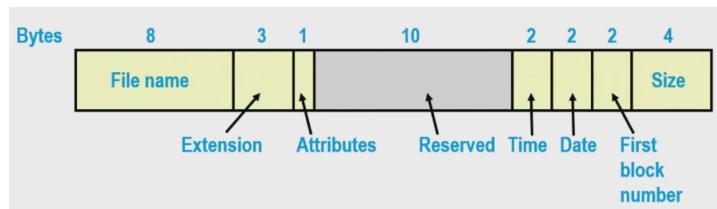
- קיובץ : חלוקה לוגית של הקבצים לפי מאפיינים או שימוש - כל התוכנות בתיקייה אחת, בתוכן כל התוכנות שכתבוו לתרגילים ב-OS וכו'.

מכאן שמערכת הפעלה היא עצם עמוק שרירוטי, כאשר כל קודקוד הוא קובץ או תיקייה (עלים קבצים, קודקודים פנימיים הם תיקיות). כל תיקיה מתאימה שם לאובייקט של קובץ, כאשר ההתאמה זו שומרה בקובץ גם היא, מה שהופך את התיקייה לקובץ בפני עצמה (שומרים בית שקובע אם הקובץ הוא תיקייה או לא).

IMPLEMENTATION OF VALIDATION

- שימוש בסיסית : מערך עם כניסה בגודל קבוע, קל למימוש אבל לא אפשר גמישות בנסיבות (לרוב כניסה יחסית קטנה כלומר מעט אטריבואטים, ואי אפשר להוסיף).

דוגמה במערכת הפעלה MS-DOS השתמשנו בשיטה כזו, ראו איור לסייע של כל כניסה רשימה כזו.

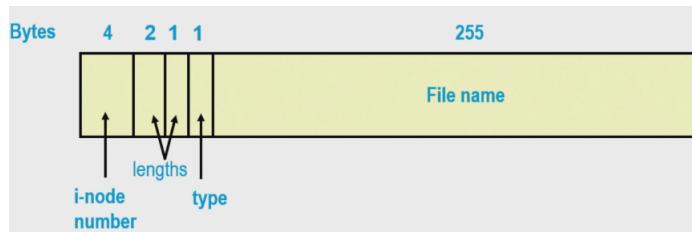


איור 71 : כניסה במערכת הקבצים של MS-DOS

- שימוש עם i-node : שימוש עם שמות ופונקטים לבני נתונים מסוג node- שומר בתוכו מספר דינמי של אטריבואטים.

דוגמה בזוניקס 7 השתמשו בשיטה כזו, אשר כניסה ברשימה היא בסה"כ שם הקובץ ופונטרא ל-node- כאמור.

דוגמה ב-ext4 (מה שמשתמשים בלינוקס כיום) משתמשים גם בשיטה זו, רק שעכשיו היא נראית כבאיר. שימוש לבניט טיפ שיעשו נמצא מחוץ ל-node-i, זה נועד לכך שפקודה ls שמדפסה מה יש בתיקייה לא תצטרך להיכנס ל-node-i של כל קובץ כדי לגלוות אם הוא באמת קובץ או תיקייה וכן חוסכים הרבה זמן של קריאה בדיסק.



איור 72 : כניסה במערכת הקבצים של ext4

ה-path של כל קובץ מתחילה מהשורש (אם הוא absolute) או ב-working directory (אם הוא relative).

הערה כדי להגיע לקובץ באמצעות path שלו, צריך שתי קריאות מהזיכרון לכל חלק של path (בין כל שני /) כי צריך לקרוא את תוכן i-node- של התיקייה ואז לקרוא את התוכן של התיקייה באמצעות הפונטרא שמצוין לשם שנמצא ב-node-i.

הערה כקונבנצייה, ה-node-i ה-0 הוא של ה-root directory (כל ה-node-iים מצויים במערך משליהם).

הגנה על קבצים וההרשאות

נרצה שמי שיצר את/הבעליים של הקובץ יוכל לקבוע מי יכול לעשות איזו פעולה על הקובץ.

הערה יצירה/מחיקת קובץ זו פועלה על התקינה עצמה.

כל כניסה למחשב קובעת id-user (המחשב במקרה הזה יכול להיות גם רשות, לא רק מכונה פיזית). כל משתמש יכול להיות חבר בקבוצות שונות, לדוגמה ghnadi AOL יהיה חבר ב-indianqms (indianqms). את הרשאות מעבר לכך אפשר לתאר בשתי דרכים:

- **מינימלית**: הפרדה באמצעות רק בעלים אחד, קבוצה אחת וכל מי שלא השניים הקודמים. כל מאוד לשמר הרשאות כך אבל אי שוב זו לא מערכת גמישה במילוי.

דוגמה בלינוקס משתמשים ב-9 ביטים לצורך לייצג הרשאות. 3 ביטים לכל אחת מהקבוצות הנ"ל. 3 הביטים לכל קבוצה מייצגים האם מותר להם לכתוב, האם מותר להם לקרוא והאם מותר להם להריץ. אפשר לייצג כל הרשאה במספר בסיסי אוקטלי, לדוגמה 110 (אפשר לקרוא ולכתוב אבל לא להריץ) מיוצג ע"י 6 (הمرة מייצגת בינארי קלאסית למשהה). בסוף, מציגים את כל הרשאות במספר תלת ספרתי (עם ספרות מ-0 עד 7).

- **מפורטת**: לכל קובץ יש רשימה שקובע איזה משתמש יכול לעשות מה. שיטה זו מאוד גמישה אבל קשה לייצוג קומפקטי.

דוגמה בוינדוס משתמשים ב-Access Control List ACL שבסך כניסה שלה יש מזהה של משתמש/קובץ, פעולות על הקובץ, האם את הפעולות אלה מותר או אסור לעשות.

אם לאובייקט אין ACL אפשר לעשות הכל, אם ה-ACL הוא null אסור לעשות כלום. אם יש כניסה (ACE), נלקUPII הפניה השולחת את המשתמש והפעולה הנוכחית, כאשר כניסה שאוסרת על פעולות מסוימות מעל אלו שמאפשרות פעולות. אם הגיעו לסוף הרשימה הרי שאסור לנו לבצע את הפעולה.

לכוארה לא צריך את האיסורים כי אפשר פשוט לא-אפשר אבל איסורים שימושיים כשרוצים הרשות לכלם חוץ מ- (נשים ACE של איסור למשתמש ספציפי, ולאחר מכן אפשר לכלום).

קישורים לקבצים

hard link • פוינטר לקובץ עצמו, קשרו אליו ואחראי עליו.

soft link • קובץ שמכיל פוינטר לקובץ אחר (קייזורי דרך בוינדוס).

הערה אם נזיז קובץ עם hard link הכל יעבד בסדר כי האובייקט עצמו לא זז. אם נזיז קובץ שיש soft link אליו, ה-link יחול עלעובד כי הפוינטר שנמצא ב-soft link כבר לא קיים באותו המקום.

פתרונות קבצים

לכארה אין סיבה לפתוח ולסגור קובץ באופן ייעודי, אפשר פשוט לקרוא ממנו כל פעם וזו. הבעיה היא שהזורך דרוש מאייתנו לפרסר את ה-path כל פעם מחדש, וכך נעדיף לעשות את זה פעם אחת (локח הרבה זמן) ומשם לקרוא באופן ישיר מהdisk בכל קרייה.

כשמדובר על PCB-ים של תהליכיים, הזכרנו שיש לכל תהליך רשיימה של קבצים פתוחים שלו. כשהתהליך מסתיים, הקבצים שלו נסגרים אוטומטית ולכך לא חייבים לקרוא ל-close בתהליך (אבל עדיף מסיבות אחרות).

כשפותחים קובץ קורים כמה דברים באופן אוטומטי:

- מפרסרים את ה-path.
- מעתקים את ה-node-i של הקובץ לטבלת הקבצים הפתוחים בכל המערכת.
- שמים פוינטר ב-PCB של התהליך הנוכחי שמצויבע ל-node-i המועתק.
- מחזירים פוינטר לכינסה ב-PCB שמתיחסת לקובץ.

מימוש מערכת הקבצים בדיסק

איך נשמר את הקבצים על הדיסק? מה מבנה הנתונים הכיטוב לכך? **צריך מבני'ת** שתומך גם בגישה סדרתית (למערכאים) וגם גישה אקראית (במסדי נתונים).

- שמירה באופן רציף של הקבצים: זה כרגע יוצר בעיה של פרוגמנטציה ולא אפשר להגדיל את הקובץ. זה לא רלוונטי ל-DVD-ים שם לא מוחקים קבצים אלא רק קוראים.

- קומפקציה: סידור חדש של הזיכרון הרציף, זו פעולה מאוד יקרה ולא משתלמת.
- בלוקים: נחלק את הדיסק לבlokים של 4KB ונותusk אותם בדומה לדפים.

כשמשתמשים בשיטה זו האבstarטרכיה מתבטאת באופן חד ביוון, מבחינת מה כל חלק במערכת רואה:

- המשמש: מידע רציף.
- ה-API: סדרת בייטים בגודל שרירותי.
- מערכת הקבצים: אוסף בלוקים (4KB).
- הדיסק: סקטוריים נפרדים (512B).

מה שדרוש מחשבה למימוש זה הפער בין ה-API למערכת הקבצים - בדומה להטעסות האינטואטיבית בין זיכרון וירטואלי לפיזי.

דוגמה `getc` ו-`cputc` שקוראים רק בית אחד, מערכת הקבצים (בגלל שהוא עובדת רק בבלוקים), תביא 4KB לזכרון ות kra'a שם בית אחד.

דוגמה קראית N בתים מקובץ (חחל מהמקום אליו אנחנו מצביעים בקובץ) ממומשת ע"י זיהוי הבלוקים הרלוונטיים, קראיהם והעתקת החלקים הרלוונטיים מתוך הבלוקים אל הבادر של המשתמש (מתקבל כפרמטר).

דוגמה כתיבת A בתים זה כבר יותר מורכב כי שכותבים לבлок צריך להיזהר לא לדרוס את מה שהיה כתוב לפני.

1. נזהה את הבלוקים הרלוונטיים.

2. אם הבלוקים לא ריקים, נקרא את הבלוקים (כדי לא לדרוס את מה שכבר שם, כי אנחנו כותבים ביחידות של בלוקים, אין לנו גישה יותר עדינה).

3. נעתק את הבادر של המשמש לבlokים buffer cache וואז נכתב את הבלוקים לדיסק.

4. נעדכן `node-i` את מספר הבלוקים, זמן אחרון שונות הקובץ ואמנו שוסף בלוקים חדשים, גם את גודל הקובץ הנוכחי.

נשמר בזיכרון את ה-`buffer cache`, אזור בזיכרון שמקורו (ע"י מערכת הפעלה, למען מערכת הקבצים) ל"מטען" של הדיסק לבlokים. כדי לאטר האם הבלוק שלנו נמצא כרגע ב-`buffer cache`, ישנו `hash table` שהמפתחות שלו הם מספר הבלוקים.

לא נסיר באפר אלא אם נדרש לגרש אחד בשביל אחד חדש. לכן גם כאן צריך אלג' גירוש רק שהפעם יוכל למש `LRU` כי בלוא הכי כל הניהול כל כך יקר ששמירה ועדכון של `linked list` של בלוקים היא זינחה ביחס לזמן שלוקחות כל שאר הפעולות.

גישה לבלוקים השיככים לקובץ

איך נדע>If הינו נמצא הבלוק על הדיסק? צריך העתקה ממספר הבלוק למיקום שלו בדיסק, וגם לזכור מה הסדר של הבלוקים בכל קובץ.

• רישימה מקושרת נאיבית: כל בלוק מכיל תוכן של הקובץ ומצבייע למיקום הבלוק הבא

לשיטתו שני חסרונות מרכזיים: גישה לבלוק שירירתי דורשת מעבר לינארי על כל הרשימה; ושבגלל שהמצבייע תופס מקום בבלוק, גודל הבלוק האפקטיבי לתוכן הקובץ הוא כבר לא חזקה של 2 וזה הורס לנו את כל החישובים.

• FAT (ממומש ב-MS-DOS): נשמר טבלה (File Allocation Table) בזיכרון שתממש לנו את הרשימה המקושרת ותשאיר את הבלוקים לתוכו בלבד.

נמשץ זאת באופן הבא: נשמר את הבלוק הראשון ב-`node-i`, ואז הערך שנמצא באינדקס הבלוק הזה בטבלה, הוא הבלוק הבא וחוזר חלילה עד שנגיע לבלוק שאחנו צריים.

כלומר לא התהמקנו מעבר על רישימה מקושרת, פשוט עכשו היה מחוץ לבלוקים עצם.

יתרונות: כל הבלוק מוקצה לתוכן הקובץ; random access הרבה יותר מהיר מרשימה מקושרת בדיסק כי זו "רישימה מקושרת" בזיכרון.

חסרונות: הטבלה מאוד גדולה עבור דיסקים גדולים ולכן צריך להביא אותה מהדיסק לזכרו אם לא יכולה בזיכרון.

• בלוקים מאונדקסים (ממומש ביונייקס): ה-`node-i` מכיל 12 פוינטורים ל-12 הבלוקים הראשונים (direct blocks), הפוינטר ה-13 מצביע לבלוק שמכיל פוינטרים לבלוקים של (תוקן) הקובץ (single indirect), הפוינטר ה-14 מצביע לבלוק שמצבייע לבלוקים שמצביעים על בלוקים של תוכן הקובץ (double indirect) ואחריו (triple indirect).

את ה-48KB הראשונים אפשר לקרוא ישיר מה-`node-i`, את ה-4MB הבאים באמצעות `single indirect`, ואז 4GB ו-4TB בהתאם לעומקם 2 ו-3. כך כל בלוק ניתן למציאה באמצעות כל היותר 3 קריאות מהדיסק.

אפשר לשמר בטען (לדוגמה `buffer cache`) את ה-`single indirect`, `double indirect` ו-`triple indirect` כל פעם לקרוא אותם מחדש.

ההיגיון מארורי זה הוא שהתפלגות גודלי הקבצים מאד מוגהה (לכיוון קבצים קטנים). באופן טיפוסי ביונייקס, חצי משטח הדיסק נתפס על ידי 0.3% מהקבצים ו- 11% מהdisk נטפס ע"י 89% מהקבצים ולהפוך (הרבה קבצים מאד קטנים).

באופן מוחשי למשמעות, 95% מהקבצים הם לכל היותר 48KB ולבן משתמשים רק ב-blocks-direct.

אם הקובץ הוא פחות מ-60 בתים (הגודל שתוופטים כל הפוינטרים הנ"ל), אפשר פשוט לשים את תוכנו במקום הפוינטרים ולהדילק בית שמכריז על הקובץ כ-*inline* (בתוך ה-node-i), זה שימושי ל-6% מהקבצים.

תרגום

התרגום הוא חזרה אינטואטיבית על מה שעשינו בהרצאה וזה שלפניה וכך לא עשה את כל הדוגמאות וההסבריםשוב. קובץ הוא Type Abstract Data Type, יש לו שם, תוכן ומטא-דאטה ואפשר לעשות עליו פעולות. זו היחידה הבסיסית שיש על הדיסק. קובץ מיוצג בתורה כניסה בתיקייה שמכילה את שם הקובץ ומצבייל ל-node-i שלו, מבנה נתוניים שמכיל את המטא-דאטה של הקובץ ומצביעים לבLOCKים שלו (שמוחלקים ל-single-indirect, double-indirect, direct blocks וכו').

דוגמה ה-node-i מצבע לבLOCK בגודל 4KB (2¹²) וכל פוינטור הוא 4 בתים (2²) במערכת 32 ביט ולכן ה-node-i יכול להכיל עוד 4MB בЛОקים.

תיקייה היא קובץ, שכן יש ל-node-i שכתוב בו בין היתר שזו תיקייה, ומצביע לתוקן התיקייה שהוא קידוד של התאמות שמות הקבצים/תיקיות ה-node-i-ים שלהם (נקרא גם מדריך).

הערה שם הקובץ לא שמור בקובץ עצמו (וגם לא ב-node-i שלו), אלא בתיקייה בה הוא נמצא.

כדי לנחל את מערכת הקבצים משתמשים ב-superblock. ה-superblock אינו בлок בדיסק, אלא איזור בזיכרון שמנהל את ההקצאה של בלוקים במערכת הפעלה. הוא מכיל את גודל מערכת הקבצים, רשימה של BLOCKS פנויים, node-i-ים שלא השתמשו בהם ועוד. בעזרתו המידע הזה אפשר להציג עוד/להחזיר בלוקים מקבצים שנמחקו וכו'.

ה-node-i-ים שמורים במערך (סופי) שמקצה כמערכת הקבצים מאותחלת, לכן מספר הקבצים שמערכת קבצים יכולה להכיל הוא סופי (אבל המכסה מאוד גבוהה אז לרוב לא מגיעים אליה). כדי לגשת ל-node-i כל שנוצר הוא האינדקס שלו במערך.

ה-node-i שומר רשימה של כמה node-i-ים פנויים לקבצים חדשים ומוסיף אליה כשמות קבצים וлокח ממנו כשיוצרים, אם צריך עוד הוא לוקח מהdisk. הכוונה היא להימנע מגישות שניתן למנוע לדיסק (מאוד יקר).

בנפרד מהרשימה הנ"ל ישנה רשימה של BLOCKS פנויים בדיסק להקצאה לכתיבה חדשות/קבצים חדשים.

ה-syscall של פתיחת קובץ מחזיר field descriptor שהוא שום דבר אבל נשלח אותו לכל syscall שנרצה להפעיל על הקובץ הזה. ה-FD הוא למעשה פעולה אינדקס ב-File Descriptor Table שהיא טבלה של קבצים פתוחים שייכים לתהילך (שמורה ב-PCB).

הערה לחוטים, מכוח כך שהם שייכים לאותו תהילך, הם בעלי אותה טבלה FD-ים.

ביוניקס כל דבר הוא קובץ (ככה המשמש רואה את זה). בכל יצירת תהליך, נפתחים שלושה קבצים אוטומטיים - Standard Input, Standard Output, Standard Error .0, 1, 2 -FD.

ברגע שנפתח קובץ, שומרים בטבלה של כל המחשב את ה-node-i של הקובץ הפתוח, שמיים פוינטר לשם ב-PCB של התהליך ומחזירים את האינדקס בטבלת ה-FD-ים שבו נמצא הפוינטר, הלא זה ה-FD של הקובץ.

הערה בטבלת הקבצים הפתוחים אנחנו שומרים גם את החיסט שלנו בתוך הקובץ וכן האם יש לנו בכלל הרשאה לקרוא את הקובץ הזה.

דוגמה נרצה לכתוב 100 בתים החל מהbite האחרון של הקובץ, שהוא בית מס' 200, כאשר כל בלוק בדיסק הוא 1024 בתים. נדרש לכתוב את 48 הבטים הראשונים של התוכן לבlok השני ואת השאר בבלוק שלישי החדש שנרצה.

בגלל שאחננו קוראים וכותבים פר-בלוק ולא ברזולוציה נמוכה יותר, נדרש להעתיק את תוכן הבלוק השני ל-buffer cache, לכתוב את הבטים החדשים בו, לכתוב חזרה לדיסק ואז להקצתו בבלוק שלישי לקובץ, לכתוב אל תוכו את הבטים האחרונים של המשמש ולכתוב את זה לדיסק. בנוסף חשוב לעדכן את ה-node-i בוגע לבlok שנוסף (וכמובן תאריך השינוי האחרון, גודל הקובץ וכו').

לסיכום, ה-node-buffer cache חשוב במיוחד לשיפור ביצועים אבל הוא יכול לגרום גם בעיות מהימנות כי בזמן שימוש מערכת הקבצים כותבת לעצמה את התוכן ב-buffer cache, יכול להיות שניתקו את המחשב ואז לא כתבו את התוכן לקובץ (לכן חשוב לא לנתק דיסק-או-קי לפני שהוא מסיים פעולה).

טבלאות במערכת הקבצים

• **File Descriptor Table** : שונה לכל תהליך, כל כניסה היא פוינטר לכינסה בטבלת הדפים הפתוחים, האינדקס הוא ערך החזרה של syscall (open).

• **Open Files Table** : שונה לכל תהליך, כל כניסה מכילה פוינטר לכינסה בטבלת ה-node-i-ים ואת החיסט הנוכחי בקובץ. יכולות להיות כמה כניסה של אותו הקובץ, אחת לכל קריאה ל-open.

כל-node-i קיימים רק פעם אחת אבל המצביעים רבים אליו מאפשרים שכל FD יהיה בהיסט אחר בקובץ.

• **i-node Table** : טבלה משותפת לכל התהליכים, כל כניסה מכילה אליון מצביעים של-node-i של קובץ קיים (פעם אחת לכל קובץ).

שאלה ענו על שאלה הבא.

6) כמה פעולות קריאה וכתיבה של בלוק מהדיסק צריך לבצע בהרצת הוכנת הבאה (הנה שהקובץ קיים אבל ריק, גודל כל מודריך בלוק אחד, ושם דבר לא נמצא בזיכרון מראש, ולא צריך לגשת לדיסק כדי להקצת בלוקים):

```
fd = open("/x/y/z/foo", O_CREATE);
write(fd, &buf, 13);
close(fd);
```

א. 5

ב. 8

ג. 11

ד. מספר אחר

איור 73 : שאלת מבחן

אנחנו פותחים קובץ עם 4 אלמנטים ב-path לא כולל הקובץ (השורש ושלוש תיקיות). נקרא את ה-node-i של השורש ואז את המדריך שלו, ואז את ה-node-i של x והמדריך שלו וכוכו', ומהישוב קצר זה לוקח 8 קריאות לדיסק. לאחר מכן כותבים לבלוק (הוא ריק) ואז בסגירה נכתבת ה-node-i-זורה ל-i-node Table, סה"כ 11.

העובדת שהקובץ קיים חשובה כי אז אומר שצריך לטען את ה-node-i של foo מהדיסק, אחרת לא היה צריך, פשוט היינו יוצרים אחד חדש.

העובדת שכל מדריך בגודל בלוק אחד חשובה כי אז קריאה של ה-path היא 2 בלוקים לכל אלמנט (בלוק אחד, לשתי קריאות שונות).

העובדת שהוא שום דבר לא נמצא בזיכרון מראש אומרת שאי אפשר להניח שחסכנו קריאות לדיסקים.

העובדת שלא צריך לגשת לדיסק כדי להקצת בלוקים חשובה כדי שההקצאה החדשה של ה-write תדרוש בלוק יחיד ולא כמה).

שאלה קראו את הנתונים הבאים.

שאלה 3: מערכות קבצים (24 נקודות)

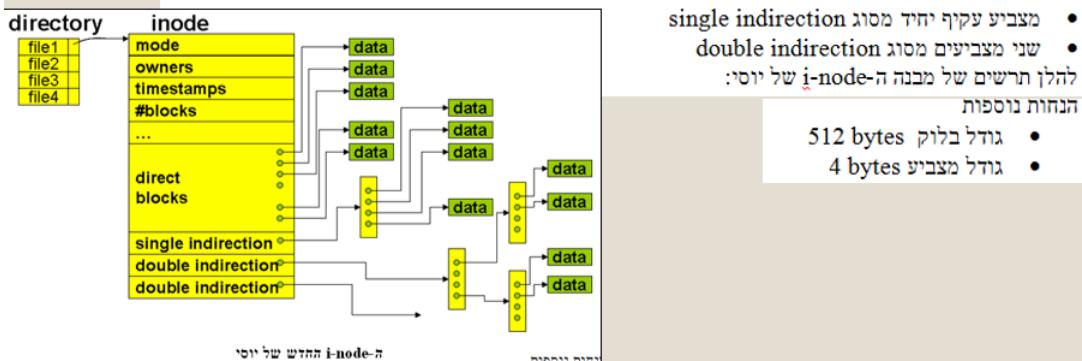
השאלה הבאה מתייחסת למבנה הנתונים **i-node** המשמש לאחסון נתונים על הדיסק, שנלמד בהרצאה על מימוש מערכות קבצים. (התרשים מההרצאה מופיע בעמוד הבא)

יוסי, הכותב את מערכת הפעלה **Yosix**, ניגש לממש מערכת הקבצים. לאחר וניהר לו מסובך לממש והעריך כי רוב הקבצים במערכת יהיו קטנים, החליט להסתפק ב-**Triple Indirection** שיבוא במקומו. בנותה, לאחר לפיכך המיפוי לבЛОקי הקובץ מה-i-node, שיציר נראה כך:

- 24 מצביעיםشيرיים לבLOCKי הקובץ ממוספרים 0- עד 23 direct
- מצביע עיקוף יחיד מסוג single indirection
- שני מצביעים מסוג double indirection

להלן תרשימים של מבנה ה-i-node של יוסי:

- גודל בלוק 512 bytes
- גודל מצביע 4 bytes



איור 74 : נתונים לשאלה מבחון

1. ה-i-node מכיל הרבה נתונים, owners, timestamps וכו', אך לא את ה-offset בקובץ. היכן הוא שומר ומדוע הוא שומר במקום מרוחק?

הוא שומר בטבלת הקבצים הפתוחים, כדי שיוכלו להיות היסטים שונים לכל פתיחה של הקובץ.

2. מהו הגודל המקסימלי של קובץ בבלוקים שניין להציגו אליו באמצעות ה-i-node כזה וכמה בЛОקים סה"כ יתפוס קובץ כזה על הדיסק (כולל index blocks אבל בלי ה-i-node)?

$$\text{הפיינטרים הישירים נוטנים ל } 2^7 \text{ בלוקים, ה-i-node } \frac{2^9}{2^2} = 2^7 \text{ וכל single indirect נוטן ל } 2^7 \text{ (} 2^7)^2 \text{ ולכ"ז } .2 \cdot 2^{14} + 2^7 + 24 = 32920$$

בנוסף לכך, צריך להתייחס למקום שתופסים הבלוקים של ה-i-indirectים. ה-i-single דורש בלוק יחיד נוסף וכל double דרוש $.32920 + 2 \cdot (2^7 + 1) + 1 = 1 + 2^7$ בלוקים (אחד לעומק 1 ועוד 2^7 מוצבים בעומק 2), כלומר סה"כ $32920 + 2 \cdot (2^7 + 1) + 1 = 32923$

3. מהו הגודל המקסימלי של קובץ שניין להציגו אליו לפי ייצוג ה-i-node שנראה בהרצאה?

чисוב דומה רק שהפעם יש 12 ישירים ו-*indirect* אחד מכל סוג (single, double, triple).

4. בספרייה /usr/yossi ישמו קובץ בשם myfile שגודלו 32KB, כמה בLOBקים יתפוס קובץ במערכת הפעלה של יוסי (כולל index blocks, לא כולל את ה-i-node)?

$$\text{אנו צריכים } 2^6 = \frac{2^{15}}{2^9} \text{ בלוקים בשביל הקובץ. הישירים נוטנים ל } 2^7 \text{ בלוקים. ה-i-single נוטן ל } 2^7 \text{ בלוקים שזה מספיק. לכן נשתמש ב-65+1 = } 65 \cdot 2^6 + 1 \text{ (אחד ל-} i\text{-block של index block)}$$

5. כמה בLOBקים יתפוס אותו קובץ בייצוג i-node שראינו בהרצאה?

אותו הדבר, כי השינוי במספר הישירים לא משפיע כי הקובץ גדול מדי.

6. נתונה קריאת המערכת: open(fd, "/usr/yossi/myfile", O_RDONLY);

(א) הנicho שהקובץ אליו ניתן לגישם קיים והוא קובץ רגיל. מה המספר המינימלי של גישות לדיסק שיתבצעו ע"י קריאה זו?

0, אם כל הנתונים כבר ב-buffer cache.

(ב) מה מספר הקריאה המינימלי של גישות דיסק שיתבצעו ע"י קריאה זו?

6 קריאות בשביל ה-path ואחת לפתחה (קריאה ה-i-node).

שאלה ענו על השאלה הבאה.

במערכת קבצים מסווג node-i (כפי שראינו בכיתה), הנicho של node-node-i מכיל 6 מצביעים ישירים ועוד שלושה מצביעים עקיפים: single indirect, double indirect, triple indirect. שימוש לב אלו מנהים שהשינוי היחיד במערכת הקבצים מושהצגה בכיה הוא במספר המצביעים. גודל כל בלוק הוא 1024 בתים (1024 bytes), גודל כל מצביע לבלוק הוא 4 בתים (32 ביט), כל מדrix (תיקיה) מאוחסן בבלוק יחיד. הקובץ c/a/b/c קיים ו אורכו 10,000 בתים. כמו כן, הנicho שמיוצג של inode Über "/" ידוע אך הוא אינו נמצא בזיכרון. נניה שבתחלת כל פעולה (בכל אחד מהפעיפים הבאים) ה-path disk cache ריק, גודלו אינסופי, וכל טבלאות הקבצים ריקות.

איור 75 : שאלה מבחון

1. האם 36 בתים מספיקים לייצוג node-i בשיטה זו?

לא! יש לנו 6 מצביעים ישירים ועוד 3 indirect, כלומר סה"כ $2^2 \cdot 9$ בתים בדiox, אבל אנחנו חייבים שdots אחרים, ולאלו אין מקום ב-36 בתים בלבד.

2. חשבו כמה גישות לדיסק נדרשת על מנת לקרוא את הבטים 2000 עד 3700 מהקובץ c/a/b/c.

כדי לפרסר את ה-path צריך 7 בלוקים. הבלוק השני נגמר ב-2048 ולכן נדרש להביא את הבלוק השני, השלישי והרביעי. לעומת זאת סה"כ 10.

אלו 4 קריאות nodes-i, 0 קריאות ל-blocks index (הכל נכנס ביישרים) ו-3 לבלוקים המכילים את תוכן הקובץ.

3. חשבו כמה גישות לדיסק נדרשת על מנת לקרוא את הבטים 7200 עד 8000 מהקובץ c/a/b/c.

ולכן צריכים צרכיים את בלוק 6 ולמקרה רק אותו כי הוא נגמר ב-8096 ולכן הוא מכיל בתוכו את כל התוכן שאנו רוצים. הישירים לא מספיקים לנו כי הם מכילים רק 6 בלוקים ואנו צריכים צרכיים את השביעי (אינדקס 6).

לכן יהיו לנו 7 קריאות לפרסור ה-path, בלוק אחד ל-single indirect ואחד לתוכן.

אלו 4 קריאות nodes-i, קריאה אחת של block index וקריאה אחת של תוכן הקובץ.

שבוע III | וירטוואלייזציה

הרצאה

פרטים ספציפיים על נישותמערכות קבצים

ראינו בהרצאה הקודמת ששאנו קוראים קבצים אנחנו שומרים את הבלוקים ב-buffer cache. אפשר להשתמש בתערובת של קבצים וניהול זיכרון. נגיד סגמנט חדש ונגיד ל-OS שמעכשו זה קובץ, עם טבלת דפים והכל, אז בשינויים למידע, תוכן הקובץ יגיע באמצעות PAGE FAULT - אם יש לנו שינויים חלק חדש של הקובץ. הסיבה שזה עוזר לנו היא שלא צריך buffer cache כל כך גדול וגם חוץ פעולות kernel כי כותבים ישיר.

דיסק הוא רכיב חומרה שמכיל טבעת מסתובבת עם יד כמו פטיפון שקוראת על הדיסק. יחידת היסוד של הדיסק היא סקטור. כדי לקרוא, צריך לסובב את הדיסק עד שהסקטור נופל מתחת לראש שקורא/כותב. אם קוראים/כותבים בסקטורים עוקבים, זה חוסך זמן סיבוב וחיפוי. בעבר ה-OS הייתה צריכה לנשל לאיזה סקטור כותבים ומתי, אבל ביום זה נעשו אוטומטיות על ידי קונטROLLER בתוך הדיסק. מערכות קבצים עוצבו כדי שמסלולי החיפוש יהיו קצרים ולא ידרשו סיבוב ותזוזה רבים. בנוסף, הזמן של כמה בקשות לדיסק דרשו אלגוריתמיקה בדומה למה שראינו, אבל כאמור הכל נעשה אוטומטית כבר.

דנו בקבצים פתוחים של כל תחילה ושמירה שלהם ל-PCB אבל לא דנו بما קורה אם כמו תהליכיים פותחים את אותו הקובץ. יש שלוש טבלאות לייצוג קבצים פתוחים כפי שהראנו בתרגול:

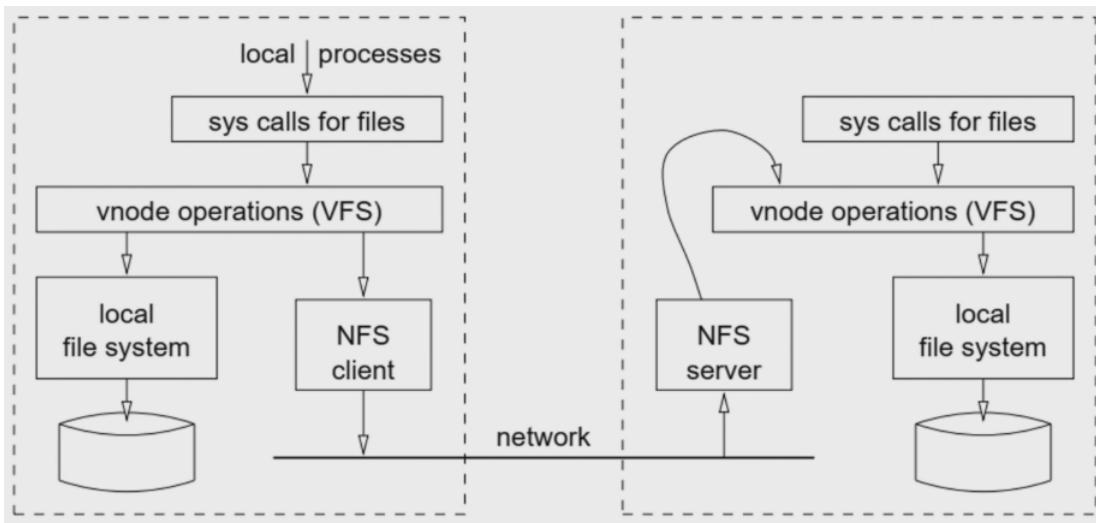
- טבלת nodes-i של קבצים פתוחים - כניסה אחת לכל קובץ פתוח לכל התהליכיים.
- טבלת קבצים פתוחים לכל המערכת - כניסה אחת לכל פתיחה של קובץ, כולל מצביע-node-i בטבלה הנ"ל ואת ההיסט בקובץ של הפתיחה זו.
- טבלת קבצים פתוחים של התחיליך - כניסה אחת לכל פתיחה של קובץ של התחיליך, מותאים File Descriptors לכניות בטבלה הנ"ל.

הערה הטעיה שהטבלה השלישייה קיימת היא שאחרת כל משתמש היה יכול לגשת למקום אקרוי בטבלה של colum ולגשת לקבצים שלא אמורים להיות לו גישה אליהם.

של מערכת קבצים היא הפיכת מערכת קבצים לשימוש בידי-OS, ובמציעים זאת עם syscall. שיטה זו מאפשרת לאחד מערכות קבצים, לדוגמה בעת חיבור דיסק-און-קי, עושים Mount לדיסק-און-קי למערכת הקבצים שלו ומקבלים אותו כ"תיקייה" בתוך מערכת הקבצים הראשית.

שיטה זו היא בשימוש נרחב, לדוגמה ב-mount מרוחק, כמו שנעשה באקווריום. כדי לעשות mount מרוחק, משתמשים ב-NFS .System Network File.

באייר הבא ניתן לראות כיצד מערכת כזו עובדת. כשמתќבל syscall שנוצע למערכת הקבצים, ה-VFS מפנה את הבקשת לגורם הרלוונטי - מערכת הקבצים המקומי אם היא מכילה את מה שאנו רוצים ואחרת ללקוח NFS, שעביר את הבקשת להאה לשרת NFS, שעביר את הבקשת למערכת הקבצים בשרת שטפלה בבקשתה, אז כל התגובה חוזרת למעלה השרת.



איור 76 : דיארגרמה לרכיבת NFS

וירטואלייזציה

המטרה שלנו היא להפריד את התוכנה מהחומרה, והתפתחות התחום הזה היא אחת ההתקדמות החשובות במערכות ממחשבים ב-25 השנה לאחריות.

עבור ארגונים, זה אפשרי Server Consolidation, כלומר במקום שיהיה שרת נפרד לכל שירות שנדרש (מייל, אתר וכו'), יש שרת אחד שמכיל בתוכו כמה שירותים וירטואלים, שכל אחד מהם חושב שהוא רץ בעצמו (בדומה לתהליכים, רק יותר מטיא), וכך חוסכים חשמל, תחזוקה וכו' כי כל שרת בנפרד לא דרוש את כל המחשבים כל הזמן.

עבור משתמשים, זה אפשרי להשתמש בשתי מערכות הפעלה באותו הזמן ולהשתמש בטובה ביותר ככל משימה.

פורמלית, וירטואלייזציה היא פרידה מהמוגבלות הפיזיות באמצעות הכוונה (indirection) שנוננת משק זהה במקרה שבו אין וירטואלייזציה (שם ההבדל לעומת אבסטרקציה).

תחת הגדרה זו, התהליך הוא Virtual Machine כפי שהאפליקציה רואה אותו. VM אמיתי הוא וירטואלייזציה כפי שה-OS רואה אותה, ככל זכרו פיזי ותרגומים כתובות, מעבד עםפקודות והרשאות, התקנים - ממש יכולו יש חומרה נוספת.

Hypervisor

השכבה שmbדילה בין החומרה האמיתית לבין הווירטואלית ומערכות הפעלה והאפליקציות שעליין נקראת hypervisor. הוא אחראי על הכוונה של החומרה הווירטואלית לאמיתית, הפרדה (חזקת) בין ה-VM-ים, Multiplexing, ניהול משאים פיזיים - כך שבסתו של דבר הוא עובד בדומה ל-OS, אבל למטרה שונה.

Popel Goldberg ו-Popel קבעו ב-1974 את התנאים לוירטואלייזציה:

- **שקלות:** VM זהה לchlוטין למוכנה פיזית, וכל OS ו/או אפליקציות ירויצו אותו הדבר.

- **בטיחות:** VM-ים מופרדים לchlוטין אחד מהשני ומהמכונה הפיזית.

- **ביצועים :** הירידה במחירות הרכזה תהיה מינורית ביחס לרכיבת על חומרה פיזית.

סוגי וירטואלייזציה

• **Multiplexing** : יצירת מספר התקנים וירטואליים על אותו המכשיר הפיזי. לדוגמה באמצעות יצירת חלוקה לדיסק, כך שהוא מתפרק ככמה דיסקים קטנים יותר ובבלתי תלויים לכל מכונה וירטואלית בנפרד. עוד דוגמה לכך היא מרחב כתובות וירטואלי, כאשר אנו מאפשרים כמה מרחבי כתובות שונים ומתרגמים אותם למרחב כתובות פיזי קטן יותר.

• **Aggregation** : יצירת התקן וירטואלי אחד על בסיס כמה התקנים פיזיים. לדוגמה ב-Raid, מערכ דיסקים עם כפליות, אשר המשק החיצוני הוא של דיסק יחיד אבל בפנים יש הרבה דיסקים שמספקים אמינות גבוהה מאוד גם אם אחד הדיסקים נכשל.

היסטוריה של וירטואלייזציה

- בשנות ה-70-60 IBM השתמשו ב-VM-ים על המיניפראים שלהם, שכל VM הרץ מערכת הפעלה שנקראת CMS.
- בשנות ה-80-90, עם המעבר למחשבים אישיים, פותחה חומרה שלא ניתן לעשות לה וירטואלייזציה, לדוגמה גישת read-only למידע רגיש.
- בשנות ה-2000 עד היום משתמשים בוירטואלייזציה לתשתיות ענן והחומרה מתאימה את עצמה גם ומוסיפה תמייה חדשה לוירטואלייזציה.

יתרונות של וירטואלייזציה

- **גיוון מערכות הפעלה** - ניתן להריץ כמה מערכות הפעלה, כל אחת טובה למשהו אחר, על אותו המחשב.
- **Server Consolidation** - יצירת גמישות ויעילות בחלוקת המשאבים וזמן המעבד שנדרש לכל רכיב בחוות מחשבים.
- **אבטחה** - הפרדה של VM-ים, ניתור התנהלות של מחשבים, לזיהוי פריצות לדוגמה.
- **נגישות** - VM-ים יוכלים לזרוץ בכל מקום ואפשר אפילו להעביר אותם למקום מסוימתם.

הערה כל ה"ל הם הבסיס של Cloud Computing. ענן מאפשר לנו לשכור VM-ים מספק ענן, שמופרדים מכל ה-VM של אנשים אחרים אבל רצים על אותה החוויה הפיזית, וכך חוסכים מעצמנו תחזוקה, כוח אדם ותשתיית מקומית.

Server Consolidation

شرطם מרכיבים, לוגית, תוכנה אחת (שרת web, שרת מייל, VFS וכו') וכן אינטואטיבית נוחיק קופסה פיזית שונה לכל אחד מהם. עם זאת, שיטה זו לא יעילה כי אף על פי שהיא מבטיחה בידוד טוב של המערכות, היא לא גמישה בכל הנוגע לחומרה, אנרגיה ומשאבים. חלק

מהמערכות לא פועלות לפעם בעוד אחריות בעומס יתר, וכל זאת באותו הזמן. לכן נרצה לחלק ביניהם את המשאבים כך שכל אחד יוכל לקבל כמה שהוא צריך בכל רגע נתון.

וירטואלייזציה מאפשרת את זה, הן מבחינות גיבוי במרקטים של קרייסה (ה-hypervisor מנטר הכל), נגשנות של השירותים, וגם בדיקה ו-deployment נוחים (הכל זו ממקום למקום די בקלות).

שימושים נוספים של וירטואלייזציה

- **תמיכה באפליקציות legacy**: גם אם אין לנו את החומרה של מחשב עתיק משנות ה-70, נוכל עדיין להשתמש ב-VM שמודה חומרה זו כדי להריץ את התוכנות העתיקות שרצנו עליה.

- **פיתוח תוכנה**: כשהתוכנה קורסת (בשלבי פיתוח וטסTING), כל המחשב יכול לKEROS. אם רק ה-VM קורס, המערכת עצמה יכולה עדיין תהיה ואפשר אפילו להשתמש בקצת מידע שמנוטר כדי לגלות מה קרה.

בפרט, כשמפתחים מערכת הפעלה, ברור מאוד למה שיטה זו מאוד נוחה לעומת פיתוח קלאסי.

בגלל-VM יכולה בתוכנה, אפשר להכניס את כולה בקובץ, כולל כל מערכת הפעלה, תוכנות, DATA, זיכרון, מצב המכשיר וכו'. קבצים אלהאפשרים לנו לעזoor את המכונה, לשמר את המצב שלו ולהמשיך ממנו הרוג בפעם אחרת, במחשב אחר (snapshots) וגם פשוט להעתיק את מערכת הפעלה לשימוש בחומרה פיזית אחרת (clones).

סוגי הייפרוייזורים

- **הייפרוייזור סוג 1 (נייטיב)** - כל החומרות הווירטואלית ומערכות ההפעלה באותו ה"גובה" - יכולים תחת שכבת וירטואלייזציה.

- **הייפרוייזור סוג 2 (hosted)** - יש מערכת הפעלה שהיא host שMRIIZAH אפליקציות עצמה, וגם מריצה הייפרוייזור שעליו יש עוד מערכות הפעלה אחרות.

- **Container** - הווירטואלייזציה היא בין מערכת הפעלה לבין האפליקציות, כך שהAPPLICATIONS חשובות שהן רצות על מערכת הפעלה אחת אבל בפועל יש אחת אחרת (דוגמא מאוד לתהיליך, רק שההכוונה היא יותר מושלמת).

היפרוייזורים בסופו של דבר מתנהגים כמו קרנל של מערכת הפעלה, כאשר VM-ים הם כמו תהליכי, הוא שולט בהכל (זמן, הקצאת משאבים). העניין הוא שמערכות ההפעלה של ה-VM-ים חשובות שהן שולטות בהכל ורצות על החומרה, ותפקידו של ההיפרוייזיר הוא לדמות זאת.

אמצעי וירטואלייזציה

- **Trap and Emulate** : שיטה שבה משתמש היפרוייזור מסוג 1, שרצ על החומרה עצמה. ה-VM רצות ב-User Mode - לא רק האפליקציות, גם ה-OS!

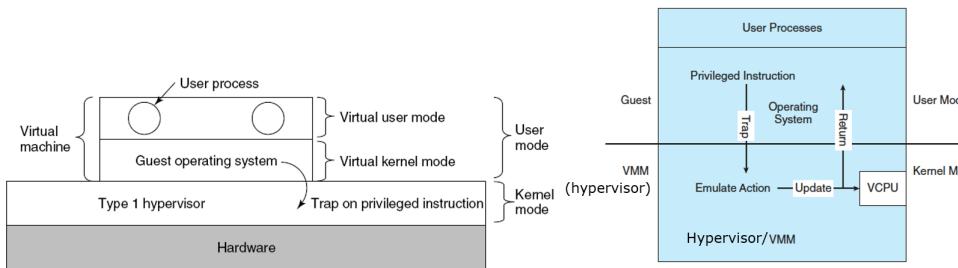
בכל פעם שה-OS מבצע,privileged instructions היפרוייזור תופס:

– syscalls וקורא להאנדר ה-OS ב-User Mode Interrupt Vector ושם שם פ' שMRIIZOT את ההאנדרים, במצב משתמש).

– פעולות privileged של ה-OS ומבצע את הפעולות בשם של מערכת הפעלה. ה-OS תנסה לעשות משהו כאילו היא במצב קרנל, קיבל exception כי היא במצב משתמש, ובמקרה לקרים נרים את מה שהיא רוצה באמצעות היפרוייזר.

שיטה זו טובה אם אין יותר מדי trap-ים, אחרת התקורה די גבוהה. ההנחה שלנו כאן היא שככל הדברים הרגיסטרים הם מפרובלגים, שזה לא היה נכון בעבר, כי ב-8x היו פעולות מפרובלגות שלא דרשו מצב קרנל.

ראו איוירים שאולי יעוזו להבין מה קרויה ויזואלית,



איור 77 : ויזואלייזות של Hypervisor סוג-1

• תרגום בינהי: שימושי אם לא כל הגישות הרגיסטר דורותות קרנל.

1. נתרגם את כל הקוד של ה-VM לקוד בטוח באופן הבא :

(א) קוד רגיל (נוחות), נשאיר אותו הדבר.

(ב) קוד רגייל, נחליף syscalls-ב-HV, שקוראות ל-`hypercalls`, שקוראים לעצמו ל-`syscall`.

2. נכניס את הקוד הזה ל-.code cache.

3. כל פעם שנריץ קוד, הוא יהיה בטוח.

• פארה-וירטוואלייזציה.

• עזרה מהחומרה.

היפרוייזרים סוג-2 (שרצים במקביל לאפליקציות על מערכת הפעלה מארחת) משאירים את רוב העבודה של התזמון למערכת הפעלה. הם גורמים ל-OS להתייחס ל-VM-ים כאילו הם תהליכי, משתמשים בקבצים של ה-OS כדי ליצג דיסקים ל-VM-ים, אפשר להרוג VM-ים עם סיג널ים וכו'.

קונטינרדים

קונטינר עושה וירטוואלייזציה למערכת הפעלה מעילה, מתוך אפליקציות. זה שימושי כ恕ץ סביבה קלה יחסית שمدמה מצבים שונים – לדוגמה כשותקנת גרסה צו או אחרית של ספריה, או קובץ קונFIGURACIA בתצורות שונות. כך ניתן לבדוק/פותח תחת מערכת שונות בלי להשתמש ב-VM שהוא כבד מדי אלא עם תקורה יחסית נמוכה, ובלי לקוחות מחשב חדש.

קונטינרים מכילים בעיקר אפליקציות, ספריות וכו' ולא מדמות את מערכת הקבצים ותזמון אלא נותנת ל-OS לעשות זאת בשביבנו. זה שונה מתחilibים בכלל namespaces ו-.cgroups

Namespaces

זה מאפיין של מערכת הפעלה, שימושה מעצבת את נקודת המבט של תחילה על המערכת. לדוגמה, תחת Namespace-ים שונים, ככל גורם לאפליקציה אחראית למספרה לקבלת גרסה אחת של הספריה, לאפליקציה אחרת שකוראת אותה הספריה (בקונטיינר אחר נגיד) לקבלת גרסה אחרת של אותה הספריה.

עם נחזר לאנלוגיה לתחilibים - תחilibים מוגדרים ע"י מרחב הכתובות וגישה מוגבלת לפעולות רגישות, בעוד קונטינרים מוגדרים ע"י ה-.namespace שלהם.

דוגמא לקונטיינר A יש namespace של PID לעצמו, כך שתחילה עם PID=1 הוא שונה לגמרי מתחילה 1 של קונטיינר B שיש לו גם namespace משלהו.

cgroups מנהלים משאים, וקובעים כמה משאב כלשהו ניתן לכל קונטיינר, בתרגום נלמד להשתמש בהם.
ראו טבלת סיכום של סוגי שונים של וירטוואלייזציות,

	Process	Container	VM
Definition	A representation of a running program.	Isolated group of processes managed by a shared kernel.	A full OS that shares host hardware via a hypervisor.
Use case	Abstraction to store state about a running process.	Creates isolated environments to run many apps.	Creates isolated environments to run many apps.
Type of OS	Same OS and distro as host,	Same kernel, but different distribution.	Multiple independent operating systems.
OS isolation	Memory space and user privileges.	Namespaces and cgroups.	Full OS isolation.
Size	Whatever user's application uses.	Images measured in MB + user's application.	Images measured in GB + user's application.
Lifecycle	Created by forking, can be long or short lived, more often short.	Runs directly on kernel with no boot process, often is short lived.	Has a boot process and is typically long lived.

איור 78 : סיכום וירטוואלייזציות

תרגול

קונטיינר עושה וירטוואלייזציה ברמת ה-OS, לעומת מכן מאפיינים שלו. עוסק בעיקר ב-Linux Container-ים, שמאפשרים ריצת כמה לינוקסים על אותו מחשב עם קרן יחיד. שני אלמנטים מרכזיים ב-LXC הם namespaces ו-.cgroups

-Namespace מאפשרים לתהליכי חישוב שהם בסביבה שונה מאשר המחשב מבחינות מאפיינים ספציפיים, לדוגמה שיש לו pid-ים שונים. ככלומר, שתהליך יחשב שהוא תהליך 1 והבא אחריו 2 וכו', במקומות שהוא יהיה 2354 ואחריו 2355 כי יש הרבה דברים אחרים. יש עוד הרבה NS-ים אחרים, כגון שם ה-host, מזהה המשמש, מערכת קבצים חדשה. כשמדוברים קונטיינר חדש, מגדרים אילו NS-ים חדשים אנחנו מגדרים.

clone היא syscall שמתקבל פ' להרצה בתהליך החדש; כתובות למיחשנות; דוגלים לתהליכי החדש הנוגעים ל-NS-ים; וארגוניים לפ' שהתהליך יירץ.

כדי ליצור NS חדש של pid-ים, נוסיף את דגל ה-**CLONE_NEWPID**.
בתרגול עצמו היו הרבה דוגמאות טכניות לתרגיל 5 שלא כוללו מטען בקורס תוכן, למעוניינים, ראו ממצגת התרגול בנושא.
אם נשתמש בדגל **CLONE_NEWNS**, **hostname**, **CLONE_NEWUTS** מカリיז על מערכת הפעלה חדשה ועוד דברים.
הערה כברירת מחדל, התהליך הבן יורש הכל מהתהליך ההורה שלו, אבל אם הדלקנו דגל ל-NS כלשהו, כל שינוי (שהתהליך יכול לעשות לעצמו) בהקשר זה ישפייע רק על עצמו (ואולי בנימים שלו) אבל לא ישפייע על האבא.

כשיצרים NS חדש pid, מבחינת האבא, הוא יחשב שה-pid של הבן שלו הוא 1. **PARENT_PID** וכיוצא"ב בינם של הבן.
כדי להשתמש במערכת קבצים חדשה צריך לעשות כמה דברים:

- להעביר את דגל ה-**CLONE_NEWNS** לדגל **clone**.
- לשנות את השורש של מערכת הקבצים לשורש של מערכת הקבצים החדשה (באמצעות **chroot**).
- לעשות **mount** לתיקייה **/proc**, שקיימת במערכת הקבצים המקורי, כדי שתופיע גם במערכת הקבצים החדשה.
החשיבות היא שלינוקס צריך תיקייה לכל תהליך שבה הוא שומר דברים רלוונטיים לתהליך, והוא עושה זאת זה ב-**proc**.
כדי לבצע את ה-**mount** הזה צריך לקרוא **mount** ורק שקsha להבין מה הפרמטרים אומרים, בתרגיל נתקבשו להשתמש בקריאה **mount("proc", "/proc", "proc", 0, 0);**
- בסיום השימוש במערכת הקבצים צריך לעשות **umount** (אצל האבא! כי execvp משתלט על הריצה וכל דבר אחריו לא רץ) באמצעות הפ' **umount (לא um, u)**.

(Control Groups) **cgroups** מגבילים את יכולות הקונטיינר החדש, לדוגמה להגביל את מספר התהליכיים שאפשר להגדיר ב-NS החדש.
כדי להגביל את מספר התהליכיים ב-NS החדש נבצע כמה שלבים:

- גדר **cgroup** חדש באמצעות יצרת התקייה **sys/fs/cgroup/pids** / בתוך מערכת הקבצים החדשה (תחת השורש החדש), באמצעות **.mkdir**.

- מערכת הפעלה תיצור אוטומטית כמה קבצים. כדי לחבר את התהיליך של הקונטינר ל-`cgroup` החדש, נכתוב את ה-`pid` שלו בקובץ `cgroup.procs` בתוך התקינה שזה עתה יצרנו.
- נכתוב בקובץ `pids.max` את מספר התהילכים המקסימלי המותר.
- כדי לאפשר שחרור משאבים בסיום, נכתוב לקובץ `notify_on_release` את הערך `1`.

פרוטוקול תקשורת הוא מערכת חוקים בין מחשבים הנוגעים לתקשורת ביניהם. לצורך זה צריכים יוכלו להודיע ולהבין הודעות מחשבים שונים, או לדעת متى לשלוח הודעות, באיזה סדר וכו'.

דוגמה כשלוחים אימיל, נרצה להעביר כתובות ומידע. אם באופן נאייתן מעביר את ההודעה כולה, בגלל מגבלות חומרה, יהיו הרבה שגיאות בשילוח ההודעה ואז ההודעה לא שווה כלום.

אפשר פשוט להגביל את אורך ההודעה, אבל זה כמובן לא פרקטי.
לכן נחלק את ההודעה לפקודות, ושלח אותן בנפרד. הבעיה עם שיטה זו היא שיכולה להיות שפקטה תעלם, או שהפקטות לא יגיעו בסדר הנכון וכו'.

לכן צריך תוכנת ביןים שתדאג לסדר ולאכוף דברים כאלה.

לרוב אין כבל ישיר בין שולח ההודעה למקבל ההודעה ולכן נדרש גם לעשות `routing` - להעביר את ההודעה בין כמה תחנות ביןים עד שנגיעו ליעד.

כדי לוודא שאין שגיאות, נוסיף כמה ביטים שבודקים שגיאות, EEC.
כל השכבות האלה ייחד יוצרות את סטאק הפרוטוקלים - רמות שונות של חוקים שצורךקיימים כדי לשולח הודעות. ראו פירוט של מודל השכבות,

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

אייר 79 : מודל השכבות

אנחנו נתמקד בשכבה ה-transport. IP דואג להעביר את ההודעות מצד לצד אבל הוא לא אמין מכל בינה אפשרית, ולכן צריך לבקר אותו איכשהו.

למד על שני פרוטוקולים, TCP ו-UDP שונים ברמת הבקרה ובדיקה השגיאות שלהם.

UDP הוא שכבה די דקה, כל מה שהוא מוסיף זה את הפרוטוקלים המקוריים של המקור והיעד (מזהה את התוכנה שלשלחה/מקבלת את ההודעה בתוך המחשב עצמו), אורך ההודעה ו-EEP.

כלומר UDP עдиין לא אמין (איבוד פקודות, אובדן סדר) ובנוסף הוא connectionless, כלומר ה הודעות בסך הכל נזירות לצד השני, בלי שבודקים שakan ההודעה הגיע וכו' .

דוגמה שכבה זו מתאימה לMKRIM שבhem הדיקט הוא לא חשוב כמו מהירות, לדוגמה משחק מחשב אונליין שדורשים עדכונים ותדרים ומהיריים למשחק מחשבים אחרים, אז אם פיספסנו משהו פעם אחת זה בסדר אבל אם לוקח הרבה זמן להודעות להגיע זה לא בסדר.

כדי לספק עוזר תקשורת אמין, הומצא TCP שהוא connection oriented, שני הצדדים מתקשרים ומסכימים על תנאי התקשרות ביניהם, שלוטרים על התקשרות במקורה של עומס, והוא דואג לאובדן פקודות ואובדן סדר וכו'. ב-TCP משתמשים בכל פעם שמהירות אינה כה חשובה, בין היתר (S)HTTP יושבים על TCP,aimail, SSH וכו' .

שבוע I/O | XIIII

הרצאה

השלמות של וירטואליזציה

נזכר שיש לנו שלושה סוגי היפרוייזר - סוג 1 שהוא על החומרה (Trap & Emulate); סוג 2 שהוא מעלה מערכת הפעלהamarah; וקווטינר שהוא מעלה מערכת הפעלה.

סוג 1 משכפל הרבה מה מה OS עשויה. סוג 2 נמנע משכפל זה ונוטן ל-OS לעשות את העבודה במקומו (היא מנהלת זיכרון, מזמן, משתמש בקבצים כדי לנהל דיסקים וכו') וכך צריך מודול בקרNEL שיעזר לוירטואליזציה (זה קל כי כל הkernelים המודולרים הם מודולרים).

במקום לעשות את העבודה במקום OS, אפשר לעשות פארה-ווירטואליזציה, כלומר לשנות את OS האורח (בסוג 2 כך שכל קריאה ל-syscall מופנת לשירותת ל-hypercall (פ' של היפרוייזר שעשו את אותו הדבר). זה הרבה יותר קל לשנות קוד מקור מאשר תרגום ביןארי וכו', אבל זה דורש קימפול מחדש של הkernel האורח. בנוסף, אי אפשר סתם להריץ OS אורח אלא צריך קוד לשנות קוד מקור וכו' .

ווירטואליזציה של זיכרון ו-O/I

איך ניתן בזיכרון וירטואלי של OS אורח, כלומר זיכרון וירטואלי וירטואלי? אי אפשר סתם לתת לכל אורח זיכרון פיזי (שהוא וירטואלי) כי אז כל ה-paging יביא לפגיעה משמעותית בביבוצים, בנגד עקרונות הווירטואליזציה. הדרך הכי פשוט לפתרון בעיה זו היא shadow

. המכונה הווירטואלית שומרת טבלת דפים כרגע, אבל גם ה-HV שומר טבלת דפים ("בצללים") שמתמחה אחר טבלת הדפים של כל VM ומתעדכנת בכל page fault (בעזרת shadow page faults כשמתגללה מיפוי של האורח שלא קיים אצל ה-HV). שיטה זו דורשת הרבה התעסוקות נוספת כמו מה לעשות עם ה-TLB.

בפרא-וירטואלייזציה, אפשר לעשות אופטימיזציות כי ה-VM יודע שהוא לא רץ *natively*. מאז 2008 במעבדים של אינטל יש תמיכת חומרה שנקראת *nested paging*, שמכירה ב-VM-ים ומאפשרת למצוא את הדף בתוך ה-VM בעילות בעזרת חומרה בלי צורך בתוכנה נוספת.

גס O/I צריך לעשות וירטואלייזציה. לשם כך יש כמה פתרונות:

- אמלציה, כלומר שה-HV עושים Trap & Emulate לדרייבר של האורח.
- פיצול, כלומר שה-HV יעשה פרא-וירטואלייזציה ויעביר את הדרייבר של האורח דרך דרייבר שלו ורק אז להתקן.
- מעבר חופשי, כלומר לאפשר לאורח לאופן ישיר עם הדרייבר בהנחה שהוא לא דורש שום גישה לפעולות רגיסטר.

התקני O/I מבחינות מערכת הפעלה

התקני O/I הם מאוד מגוונים, יש רכיבי תקשורת רק עם קלט, רק פלט, גם וגם, התקני אחסון עם קלט ופלט או רק קלט (CD-ROM) ועוד ועוד. כל רכיב שונה ועובד אחרת וצריך אייכשו לסדר את העניינים.

כדי לסדר את הכל משתמשים בסעיפים שונים מחובר מצד אחד למעבד, ומצד שני מחובר לكونטROLרים שרצים על מיקרו-מעבדים בתוך התקן שיוודעים לתקשר באמצעות פרוטוקול כלשהו עם המעבד דרך הבאס. כדי לתקשר עם הקונטROLרים, ה-OS משתמש בדריברים, שמכירים את התקנים וידעו איך לדבר עם הקונטROLרים.

מטרות מערכת הפעלה ל-O/I

- להציג אבסטרקציה לוגית של התקנים (להסתיר פרטים על משקל החומרה, טיפול בשגיאות).
- לתמוך בשימוש יעיל בהתקנים (multiprogramming).
- לתמוך בשיטות של התקנים בין תהליכי/משתמשים (ספק את ההגנות הנדרשות, לתזמון מניעה הדדית על אותו התקן).

דריברים וקונטROLרים

קונטROLרים מבחינת ה-OS הם חומרה. לרוב להתקני O/I יש רכיב מכני ורכיב אלקטרוני והאחרון הוא הקונטROLר/adapter. המשך בין הקונטROLר לרכיב המכני הוא מאוד low-level.

דוגמה בדיסק, הDATA מגע כסקטור, רצף ביטים ו-ECC והקונטROLר צריך להמיר את זה לבlok כדי שמערכת הפעלה תדע מה לעשות עם זה (כפי שלמדנו במערכת הקבצים).

לאחר מכן, דרייבר המתאים לכל התקן מדבר עם הקונטROLר כשהוא במצב קרנל, והוא מציג API אחד.

דוגמה בדיסק, לא משנה איזה דיסק זה, נקלט `close`, `write`, `open` וכו'.

מי שכותב את הדרייבר זה מי שמייצר את התחן במטרה שימושו כמה שיותר בהתקן.

הערה בגלל שהדרייבר רץ במצב קרנל, אם הוא לא כתוב היטב או כתוב באופן זמני, הוא עלול להפיל את כל המחשב.

הדרייבר מקבל פקודה מאפליקציה (כתיבה לדיסק נגיד), מדבר עם הקונטROLLER באמצעות רגיסטרים, מקבל תשובה ומוחזיר אותה. כדי לכתוב לרגיסטרים של הקונטROLLER (שהם "הารגומנטים" שהוא מקבל בשביל בקשה מהדרייבר), אפשר או לכתוב שירות ל-`I/O` או למפות כתובות וירטואליות לריגיסטרים אלה ואז להשתמש בפקודות כתיבה קלאסיות לזיירון. מעבר המידע הזה קורה כמובן דרך הbas.

דוגמה Parallel Port הוא קוונטטור עם 25 פינים שמחבר התקני `I/O` כמו מדפסות. היו 24 כבלים ואחד הארקה (למעשה 8 להארקה). הcabלים

היו

<code>D₇</code>	<code>D₆</code>	<code>D₅</code>	<code>D₄</code>	<code>D₃</code>	<code>D₂</code>	<code>D₁</code>	<code>D₀</code>
read/write data register (port 0x378)							
<code>BSY</code>	<code>ACK</code>	<code>PAP</code>	<code>OFON</code>	<code>ERR</code>	-	-	-
read-only status register (port 0x379)							
-	-	-	<code>IRQ</code>	<code>DSL</code>	<code>INI</code>	<code>ALF</code>	<code>STR</code>
read/write control register (port 0x37a)							

איור 80 : סידור הcabלים ב-Parallel Port

כאשר אנו רואים כאן בשורה העליונה את הריגיסטרים לקריאה/כתיבה שהתחברו לפורט מסויים (0x378), ביטים לקריאה בלבד (אם התחן עסוק, האם קיבל את הבקשה וכו') בפורט אחר ואז ביטים לבקרה כמו IRQ שאחראי על אינטראפטים.

כדי לשЛОח בית אחד של נתונים, הדרייבר השתמש בקוד הבא,

```

void
sendbyte(uint8_t byte)
{
    /* Wait until BSY bit is 1. */
    while ((inb (0x379) & 0x80) == 0)
        delay ();

    /* Put the byte we wish to send on pins D7-0. */
    outb (0x378, byte);

    /* Pulse STR (strobe) line to inform the printer
     * that a byte is available */
    uint8_t ctrlval = inb (0x37a);
    outb (0x37a, ctrlval | 0x01);
    delay ();
    outb (0x37a, ctrlval);
}

```

איור 81 : קוד דרייבר של Parallel Port

קודם כל אנחנו בודקים שההתקן לא עסוק (busy wait, מכאן בא השם), שמיים את הבית בפינים 0 עד 7 (כל אחד בית אחד כמובן), ומתריעים לكونטROLLER ע"י הדלקת בית ה-STR (strobe) להציג מיקרו שנייה ואז החזרתו לערכו המקורי.

כיום רוב התקנים מתחברים דרכם USB שמאפשר ארבעה סוגי תקשורת:

- אינטראפט - קצת מידע ש מגיע מההתקן, בשביל עכבר או מקלט.
- העברת 64 בתים עם ECC בכל פעם, בשביל מדפסות וstoriks.
- Isocronous - סטרים נתונים בלי ECC (אודיו, אוזניות).
- בקרה - ניהול התקנים וכו'.

USB מאפשר להכניס ולהוציא התקנים בזמן שהמערכת רצה, ושומר מקום ברוחב הפס שלו להתקני אינטראפט ו-isochronous (עד 90%) והשאר נשאר לבאלק ובקרה. הודעות הן בגודל של 1500 בתים, ובתוכו חילוקה בין התקנים.

שיטות מעבר מידע

- Polling - המעבד אחראי על מעבר המידע, זיהוי סיום הפעולה של ההתקן. הוא עושה זאת ע"י בדיקה רציפה וחזרת של הרגיסטרים של הקונטROLLER (busy wait הוא דוגמה לכך).
- אינטראפטים - המעבד אחראי על מעבר המידע אבל הקונטROLLER אחראי על התראעה למעבד על סיום הפעולה באמצעות אינטראפט. שיטה זו מאפשרת multiprogramming

- DMA - המעבד מתחילה את הפעולה, ה-DMA מעביר את המידע מהזיכרון לרегистרים של הקונטROLLER, ומעלה אינטראפט כמעבר המידע נגמר.

דוגמה נראה איך זה קורה במקרה של דיסק :

1. הדרייבר מתבקש להעביר בתים מהדיסק לבארט בזיכרון X .
2. הדרייבר אומר לkontroller להעביר C בתים מהדיסק לבארט.
3. kontroller מתחילה לדבר עם ה-DMA.
4. kontroller שולח כל בית ל-DMA.
5. ה-DMA כותב לבארט שמתחליל ב- X כשהוא מוריד את הערך של C עד שהוא 0.
6. כאשר DMA, $C=0$, DMA עושה אינטראפט למעבד.

הערה DMA נלחם עם המעבד על גישה ל זיכרון, אבל זו לא כזו בעיה כי המעבד לרוב לא נוגע בזיכרון כי יש לו מטמון.

הערה לאורך השנים, הkernel של Linux גדול מאוד, כאשר יותר מחצית ממנו הוא דרייברים.

שכבות ביןיהם

יש שכבות ביןיהם בין קריאות ה-API של המשתמש לבין הדרייברים שמחלקת את התקנים לשולשה סוגי התקנים :

- .read/write ו open/close - Block Oriented
- .read/write ו open/close ו get/put - Stream Oriented
- .send/receive ו read/write ,open/close - Network Oriented

מערכת קבצים משתמשת עד שכבתם בין האפליקציות לממשקים הנ"ל כי הפעולות על התקנים דומות מאוד לאלו על קבצים.

תוספות לתקשורת עם התקנים

בכל התקנים משתמשים בכמה שיטות לשיפור ביצועים/נוחות :

- Buffering - מאפשר פעולה א-סינכרונית של יוצרים-צרכים, מאפשר גדלים שונים של מידע ואפשר להיפטר מיצרך/צרוך בזיכרון ולשמר אותו בדיסק עד שימושו מלא את הבארט.

דוגמה באפר ציקלי בזיכרון-צרוך שראינו בזמןנו היא דוגמה לבאפרינג.

- תיקון שגיאות - מאפשר לוודא תקינות של תוכן, באמצעות זיהוי שגיאות וכן תיקון שגיאות.

מבוא לאבטחה

עד כה בכל הנוגע לאיכות המערכת, התעסקנו באמינותה המערכת, כלומר שלא יקרו טוויות - הימנעות מDDR-LOCK, מניעת קריישה של המחשב אם המשמש עושה משהו מוטומט. מערכת מאובטחת היא בטוחה מפני מתקפות זדוניות. זה כולל גם למנוע שימוש לשולוט ולהשתמש לרעה במחשב, וגם מתקפות מכונות ומותוחכבות.

מבחינת מערכת הפעלה, אבטחה מתחלקת לשני חלקים :

- לפרוץ לחשבון של המשתמש ומשם לנשط לקבצים רגיסטרים ולהתחזות לאדם.
- הפרוץ למערכת ולקבל גישה לכל החשבונות, ליצור חשבון חדשים, להתקין וירוסים ולרוץ במצב קריל ומשם גם לתקוף פעם נוספת.

הערה אנחנו עוסקים בחלק השני, לא הראשוני כי הוא לא מעניין.

דוגמה אפליקציית אלקטור ששמירה את מאגר הבוחרים של מדינת ישראל הכליה את הסיסמאות של האדמינים ב-.html וזו לא פריצה למערכת, אלא גישה לחשבונות.

תרגול

אנחנו עוסקים בסוקטים, שמאפשרים תקשורת בין אפליקציות. בעבר דיברנו על ה-Transport Layer (TCP) או מהירות מעבר (UDP). עתה נעסוק בתשתיית הזו - סוקטים.

סוקט הוא שער לעולם הגדול, שבאמצעתו מעבירים מידע ומקבלים מידע, והוא נוצר במפורש ע"י האפליקציות. מערכת הפעלה חושפת לאפליקציות API שככל פ' כמו send/receive וכו'.

תכנות סוקטים לפרטוקול TCP בפרדיגמת

- שרת תמיד רץ והוא צריך לפתח סוקט שפותח לחברו של משתמש.
- המשתמש צריך לפתח סוקט במחשב שלו.
- המשתמש מצין את כתובת ה-IP ומספר הפורט (סוקט) של השרת (באנלוגיה לבני אדם, כתובת ומספר דירה).
- לאחר יצירת הסוקט, המשתמש צריך להתחיל לחברו עם שרת ה-TCP.
- כשהשרת מקבל בקשה מהמשתמש, הוא פותח סוקט חדש כדי לתקשר עם הלוקה הספציפי זהה (הפורט הוא אותו הדבר אבל הסוקט שונה לכל לקוח).

הגדשה סטרים הוא רצף מידע שנכנס או יוצא מתחילה, ובהתאם סטרים פלט וקלט הם או יוצאים או נכנסים בהתאם.

דוגמה ב-TCP, משתמש יש סטרים קלט מהשרת וסטרים פלט לשילוח הودעות לשרת.

אינטראקציית סוקטים ב-TCP

- השרת פותח סוקט שמקבל בקשות לחיבורים.
- השרת מחייב להודעות מהסוקט. כשהוא מקבל, עושם "לחיצת ידיים" של TCP ואז גם הלוקו יוצר סוקט.
- הלוקו שולח בקשה מהסוקט שלו, היא מתקבלת בסוקט של השרת (החדש שנפתח לו), הוא מגיב ושולח חזרה תשובה.
- הסוקטים נסגרים.

ב-UDP אין את לחיצתידיים והשרת פשוט שולח הודעות באופן (כמעט) בלתי מבוקר לлокו.

אינטרקציה סוקטים ב-UDP

- השרת פותח סוקט לביקשות.
- הלוקו פותח סוקט ושולח דרכו בקשה שמכילה את ה-IP והפורט שלו.
- השרת מקבל את הודעה, כותב תשובה כשהוא שולח אותה לפרטים שקיבל מהлокו.
- הלוקו מקבל את התשובה, וסוגר את הסוקט.

בתרגול עצמו היו הרבה דוגמאות טכניות לתרגיל 5 שלא כולל מיטעמי בקרת תוכן, למעוניינים, ראו ממצגת התרגול בנושא.

כשנתכנת תקשורת צו, נשמר מבני'ת שמכיל סוג כתובות (לצורך עניינו IP ופורט אבל יכול להיות אחרים) ותוכן הכתובה.

כדי לכתוב בית לזכרו אפשר כתוב ב-big-endian, כלומר שהבית הכי גדול הוא משמאלי (שבו משתמשים בתקשורת) או ב-little-endian, Little Endian Big Endian, כלומר שהבית הכי קטן הוא משמאלי (פופולרי בקרב כל דבר שהוא לא תקשורת) ולכן נדרש המרה ביניהם.

- יש פ' שמצוות המרת htons, משפחת htons* וממשוואות htons-to-Host (כמו בכל שאר הפ', לא אציג את התיעוד המלא של הפ' מטעמי חוסר עניין לציבור כי זה לא מהותי).
- כדי להמיר מחרוזת IP לbytes struct אפשר להשתמש בפ' .inet_aton.
- כדי לקבל את כתובות ה-IP של הצד השני של סוקט אצלו, יש את הפ' .getpeername.

ה-DNS הוא שירות שמיר כתובות שקריות לבני אדם (host names) לכתובות IP (כך מומר google.com ל-8.8.8.8 או מה שזה לא יהיה). הפ' gethostname מחזירה את שם host שمبرיך את התכנית.

אפשר להשתמש ב-DNS באמצעות הפ' gethostbyname struct עם הרבה שדות שעוזרים.

הערה כקונבנצייה, ה-DNS יתרגם את localhost ל-127.0.0.1 שהוא תמיד המחשב שלנו.

תכנות תקשורת שרת-локו

צד השרת יראה כך.

1. יצרת סוקט באמצעות הפ' socket לצורכי סטרים או לצורכי דאטאגראם (connectionless) באמצעות אחד הפרמטרים לפ'.
2. חיבור הסוקט לכתובת באמצעות הפ' bind כך שידעו למי לפנות (נותנים לו שם לצורך העניין).
3. האזנה על הסוקט באמצעות הפ' listen כדי לקבל הודעות נכונות. כשמתקבלת בקשה לחבר בסוקט, ניצור חיבור עם הלקוח באמצעות הפ' establish (שנכתוב בעצמו, שמגלה מי התłącz אלינו, מי אנחנו, יוצרת סוקט חדש ומזינה עליו).
4. המתנה להודעות באמצעות הפ' accept בסוקט הנוכחי, שמקפיא את החוט עד שמתתקבלת הודעה.

צד הלקוח יראה כך.

1. ניצור סוקט עם .socket
2. נחבר אותו לכתובת עם .bind
3. נתחבר לכתובת והפורט של השרת עם .connect

כדי לקרוא הודעות צריך להשתמש ב-read וرك שצורך לעשות לולה שקוראת עוד ועוד בתים אל תוך באפר כי read מביא רק בתים שקיים, ואם רוצים להוציאות ל-ת' בתים צריך לעשות לולה שקוראת וקוראת עד שקרה ת' בתים.
שרת לרוב מתקשר עם יותר מסוקט אחד (File Descriptors) כי סוקטים ממומשים כאבסטראקטיה קבצים) ואז צריך להתעסק עם כמה בו זמן. אפשר להשתמש בחוטים אבל זו התעסקות וחלופין יש את הפ' select.
select מקבל אוסף FD-ים לקריאה, אוסף FD-ים לכתיבה ו-FD-ים למיון אקספנסים. היא חוסמת את החוט שקרה לה ובודקת (בטווח זמן כלשהו) האם יש FD שמוכן לכתיבה/קריאה/מיון, ומהזירה את מספר ה-FD-ים שמוכנים לפעולה. בנוסף, היא עורכת את אוסף ה-FD-ים כך שרק אלו שמוכנים לפעולה יהיו שם ואז אפשר לבדוק אם FD נשאר באוסף, אם כן הוא מוכן ואפשר לעבוד אליו ואם לא אז לא.

שבוע VII | אבטחה

הרצאה

כאמור אנחנו עוסקים בפריצות למערכת, לא השגה חיצונית של פרטיים של משתמש.

אימות

משתמשים מיוצגים ע"י חשבונות שמותאים ל-s' User ID. לכל פונקציית הרשות ספציפיות, כאשר לאדמין יש הרשות על (root).

תהליך ה登入 הוא די פשוט :

- המשתמש מכניס שם משתמש.

- המשמש מכניס סיסמה.
- המערכת בודקת את הסיסמה מול הסיסמה השמורה לשם המשמש.
- אם יש התאמה, המשמש בפנים.

הערה חשוב להצפן את הסיסמאות כדי שבמקרה של דליפה לא יהיה כל קל לדעת מה הסיסמאות של המשתמשים (עוד על זה בהמשך).

פריצת סיסמאות

יש כמה גישות למתתקפות על סיסמאות:

- ברוט-פורס: ננסה את כל הסיסמאות האפשריות עד שימושו עובד.
- ניתן לבצע מתתקפה זו offline אם כבר דלפו הסיסמאות המוצפנות. המתתקפה עצמה תליה באורך הרצוי של סיסמה ובאפשרויות לתווים (אותיות, מספרים). זמן החישוב הוא אקספוננציאלי באורך הסיסמה (בහנה ש策יך אותיות ולא רק ספרות).
- nichosh סיסמאות: נחש באופן מושכל את הסיסמה של האנשים.
- יש הרבה מאוד סיסמאות שכיחות כמו 123456, תאריכים ומיללים ספציפיים.
- מעבר לכך, הרשותות החברתיות חשופות הרבה מידע על כל העולם כך שאם בחרנו את שם החתול שלנו כסיסמה, קל לפזר לחשבון שלנו.
- בנוסף, יש תבניות מקובלות לסיסמאות, כמו הוספת ספירה בסוף למילה שכיחה זהה גם מכך על nichosh מושכל.
- גישה משולבת: לעיתים ברוט-פורס על מיליון מיללים שמכיל סיסמאות שהודלפו בעבר.

דוגמא ב-2012 פרצו לאימייל של מיט רומני, מועמד לנשיאות רפובליקאי. עשו זאת באמצעות security question של שם חיית המחמד האהובה עליו, שהייתה ידועה כי פורסם בתקשורת לפני שהוא התעלל בכלב שלו, seamus, לכאורה.

ל nichosh סיסמאות יש כמה יתרונות, ביניהם בני אדם נכנים לבניית מבנים. אפשר לבחור ביטוי של מעלה 7 מיללים אקראיות, או לחלופין להשתמש בסיסמה אקראית ארוכה ולשמור אותה ב-Password Manager (שהוא מוגן עם סיסמה מאוד חזקה).

מה מערכת ההפעלה יכולה לעשות? אם נציב מגבלות על הסיסמאות זה משחק אינסופי של חתול ועכבר.

הפתרון שבו לרוב משתמשים הוא לשים salt על כל סיסמה (עורתי לכתוב את **הערך בוויקיפדיה בנושא**). salt הוא תוספת אקראייה לכל סיסמה שמצוידים אליה לפני שמצוינים (עשויים לה hash). שיטה זו מונעת מפץ רוחבית לכל הסיסמאות כי הסיסמה המוצפנת היא לא 123456 אלא 123456fjfdb; vskb.

בנוסף, אם ההצפנה חזקה (איתית בכוונה), אנחנו מאמינים את קצב החישוב של הפוך באופן מלאכותי כדי שזה לא יהיה פרקטטי.

מעבר לכך מבחינת חווית המשתמש אפשר להציג את מספר הלוגינים האפשריים לאותו חשבון, או להשתמש ב-*biometrics*.

הערה גם כשנכנסים למחשב, חשוב להגן עם הראות גבוהה על קובץ הסיסמאות (אין סיבה שכולם יכולים לקרוא אותו, רק root).

הרשאות

במערכת הקבצים, לכל משתמש יש הרשאות ביחס לכל קובץ ותיקייה (אם הוא יכול לקרוא, להריץ). אפשר להסתכל על זה הפוך - לכל קובץ ותיקייה, מה היכולות שיש לכל משתמש (זה לא בשימוש).

בຍוניקס אפשר להשתמש ב-setuid כדי להריץ תכנית עם הרשאות של הבעלים או sudo שמאפשר לרוץ להשתמש אחר.

דוגמא כדי לעשות ping ולשלוח פאקטות, צריך להשתמש בהרשאות root.

משמעותם על מערכת חשוב להבדיל בין שני סוגים מרכזיים :

• Desktop - מחשבים עם כמה משתמשים, מעט התקנים חיצוניים משותפים, עם מידע פרטי ותוכנה מבוקרת. מתקפות מנסות להגעה לקבצים.

• Mobile - משתמש אחד, הרבה התקנים חיצוניים (מצלמה, GPS) ותוכנות הן אפליקציות מותקנות. אנחנו מנסים להגן מפני קבלת גישה זדונית לקבצים אבל גם להתקנים (הסינים רוצים לראות דרך המצלמה שלנו בכל רגע נתון).

מתקפות מתחלקות לשני חלקים מרכזיים :

• סוס ט羅יאני - תוכנה שمبرעת פעולה שעוזרת, אבל מכילה דלת אחורייה.

• וירוס - תוכנית שזוהמתה עם קוד זדוני.

כדי להגן מפני הנ"ל אפשר לעשות כמה דברים :

• אנטי-ווירוס - תוכנות שמזהותות תבניות של קוד זדוני.

• ניטור - מעקב אחר פעילות בניסיון לזהות התנהלות חשודה.

• Firewall - חוסמת תקשורת חשודה מבוחץ.

מפגעי אבטחה

• אי-בדיקה קלט : מקור מרכזי של מפגעי אבטחה הוא לא לבדוק את קלט המשתמש כראוי (שהוא עומד בפורמט הנדרש). קלט כזה יכול להכניס קוד זדוני ואז האפליקציה יכולה בטעות להריץ אותו ומערכת הפעלה לא יכולה למנוע זאת.

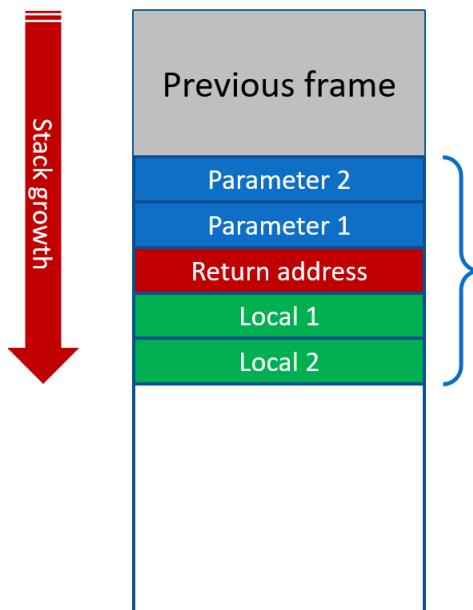
דוגמא לפניה חודשים נחפה החולשה ב-j4Log, Open Source Java. זו ספריה לוגים ל-Java. זו ספריה Open Source בשימוש מאוד נרחב שכפיכך אפשרה לקבל הודעות מותאמות מהאינטרנט, ובמקרה כזה הוא בעל בעיוורון את הקלט ואיפשר הרצת קוד במחשב של הקורבן .(RCE)

זו מתקפת zero-day ובעצם גם zero-click (מיישחו בשורת מיינקראפט שלח איזה סטרינג מזר שגורם למחשב שלנו להתחיל לבצע ביטוקין בשליל הממשלה הצפו-קוריאנית).

- **באפר-אוברפלואו :** כתיבה למערך מעבר למותר ודרישת ערכים סמוכים.

משם אפשר או סתם לדחוס ערכים ואז זה סתם באג, או לשים ערכים חדשים באופן זדוני כדי לגרום לתכנית לתקוף את המחשב. שיטה זו נקראת Stack Smashing.

קורה כי תחיליך רץ עם הרשאות של משתמש, מקבל קלט חיצוני, מעתיק אותו לבאפר של התחליך, ערכים נדרסים וכוחזרים לפ' שקרה לפ' הנוכחית, קופצים לקוד זדוני.
כך נראה המחסנית בעת קריאה לפ',



איור 28 : מחסנית בעת קריאה לפונקציה

כאשר הכתובות עלות כלפי מעלה (המחסנית מתחילה מהסוף). אם נכתב יותר מדי על local, נגיע ל-return address. אם סתם נתנו מחרוזת ארוכה מדי הסıcıוי שהכתובת הזו תהיה למקום עם קוד חוקי זניחה. אם אנחנו בוחרים באופן זדוני لأن לחזור (לדוגמא לבאפר שמכיל פקודות אסמבלי שלנו), הצלחנו לתקוף את המערכת!

דוגמה אם נצליח לשים פקודות אסמבלי כדי להיכנס לטרמינל (shell) ומשם אפשר לעשות מה שרק רוצים.

שיטה זו לא תמיד פשוטה, אבל יש דרכים להעלות את הסיכוי להצלחה. לדוגמא, לדروس כמה ערכים לפני וכמה ערכים אחרי כתובות החזרה עם מצביע לקוד הזדוני, ולשים בקוד הזדוני NOP slide, כלומר כמה פקודות שלא עושים שום דבר באסמבלי כדי שגם אם אמ hooked הפעילה קצת פיספה את הקוד הזדוני שלנו, היא תגיעה אליו עדיין.

כדי להגן מפני באפר-אוברפלואו יש כמה שיטות:

– **ערכי קנריות :** נציב ערך אקריא ממש לפני ערך ההחזרה ושמור את הערך הזה בצד. אם תוקף ידרוס את ה-return address הוא יצטרך לדروس גם את הקנריות ולפניהם שנקפוץ חזרה לפ' הקוראת, נבדוק אם ערך הקנריות לא השתנה, אם הוא השתנה אנחנו יודעים שנתקפנו.

- הערה ערכי קנרית לא יעזרו אם יש לנו דרך לקרוא את ערך הקנרית לפני (אם יש שני באפר-אוביפלואום לדוגמה).
- ASLR : נסיט את המחסנית קדימה באקראיות כך שהיא יותר קשה לנחש כתובות במחסנית (זו הסיבה שהיינו צריכים את translate_address בתרגיל 2).

- הכנסת אקרים לערכיהם של פוינטרים (הזות סגמנטים וכו').
 - למנוע הרצה של כל קוד מחוץ לSEGMENT הקוד.
- שיטה זו אינה אבסולוטית כי אפשר לקרוא לפ' libc שכתובה כו בסegment הקוד (אבל את זה מונעים עם ASLR).
- אחרת הפ' ב-libc היא system shell code-הגהה ל-.

בשוד מورد מהאינטרנט (או אפליקציות בטלפון), נרים את הקוד עם הרשות מינימלית כי אנחנו לא יודעים אם הקוד הוא זמני או לא. ככלומר, נעשה וירטואלייזציה ונגביל את יכולות של האפליקציה ל-sandbox, לדוגמה לא ניתן להריץ syscall-ים מסוימים. אנדרואיד יוצרם sandbox ע"י מתן pid שונה לכל אפליקציה, שרצה כתהילך נפרד מהאחרות ואז עם הגבלות מוכרות על ההרשאות של ה-user. האפליקציה לא יכולה לעשות שום דבר רע (לרבות מוגבלות על syscall-ים).

תרגול

נסкор בקורס את רוב הנושאים שלמדו במהלך הסטטוס

- **אינטראפטים** : אינטראפטים מתחלקים פנימיים (אקספנסים, שגיאות של המעבד עצמו) וחיצוניים (מחומרה, שטוריעים על אירוע של החומרה). אינטראפט יכול להיות בר-מייסוק (אפשר לחסום אותו) או שאי אפשר למסק אותו (לדוגמה SIGTERM - חיבים להתעסוק אליו).

אינטראפטים פנימיים קוראים אם יש חלקה לאפס, segmentation fault וכו', אבל גם יש trap PAGE FAULT או trap (כלומר לא רק שגיאות. ניתן לחלק אותם :

- aborts (שתי הדוגמאות הראשונות) שכטוואה מהם צריך להפסיק את ריצת התהיליך.
- traps שאינם שגיאה, אלא פשוט בקשה לפעולת רגישה, כאמור syscall traps. דורותים הרצת קוד של מערכת הפעלה וכניסה למצב קרnl (במציאות שניוי bit-mode). בתוך מצב קרnl, הנדר ל-trap נקרא כדי לטפל בבקשת. כשהקרnl מסיים, הוא חוזר לפקודה הבאה אחרי ה-trap בקוד התהיליך.

דוגמה התכנית מבקשת כתובת שלא ממופת בזיכרון הפיזי. קורה אקספשן SMBIOS מה-OS להביא את המידע מהdisk ולטעון אותו לזכרון. התכנית מקבלת את המידע בלי לדעת שהייתה שגיאה בכלל.

כדי לטפל באינטראפט פנימי מבצעים את השלבים הבאים :

1. שומרים את המצב הנוכחי.
2. משלטלים על הריצה ומטפלים בבקשת.
3. מחזירים את המצב הקודם.
4. מחזירים את השליטה לתהיליך.

- סיגנלים: סיגנלים נשלחים ע"י מערכת הפעלה, אפ"ג שתהליכיים יכולים לבקש מה-OS באופן רצוני לשЛОוח סיגנל לתהליכיים אחרים או לעצם (ה-OS מתווכת תמיד).

סיגנלים שונים מאינטראפטים כי הם נוצרים ע"י ה-OS בוגוד לאחרוניים שנוצרים ע"י החומרה. בנוסף, הראשונים מנווהלים ע"י התהליך והאחרונים מנווהלים ע"י מערכת הפעלה.

ברגע שמתקבל סיגנל ה-OS מעבירה אותו מיד להנדLER המתאים לה לא משנה מה, וחזרת למקום שבו הפרעה כשהנדLER מסיים.

- PCB: לכל תהליך יש PCB ששומר מידע עליו כמו ערכי הרегистרים שלו ומיקום הזיכרונו שלו (הكونטקט שלו), הקבצים הפתוחים שלו, וגם מידע חשוב למערכת הפעלה כמו priority בתזמון שבבסיס על כמה זמן התהליך רץ וכו'.
- בנוסף, נשמרים ההוראות של התהליך (איזה משתמש מריץ אותו), ועוד מידע "חשבונאות".

- חותמים: חוטים באו כדי לשפר ביצועים כשיש כמה ליבות, והם מהווים "מסלולים עצמאיים" בתכנית ויתרונם הוא שהם קלילים - עם זיכרונו משותף לחוטים אחרים בתהליך (מלבד המחסנית).

- חוטים ברמת הkernel: מערכת הפעלה יודעת שיש חוטים שונים ויכולת להריץ חוטים שונים בו בזמןית במעבדים שונים.

- חוטים ברמת המשתמש: מערכת הפעלה רואה חוט אחד והשתמש אחראי על תזמון של החוטים בפנים ויכול לדאוג שתתקיים מניעה הדדית (שהוא לא יכול בחוטים ברמת הkernel בכזו קלota). היתרונו על פני חוטי kernel הוא התקורה הנמוכה שכורוכה במעבר בין חוטים (לא צריך מצב kernel וכו').

- בעיית האזוזן הקרייטי: הרבה תהליכיים רוצים לגשת לאותו משאב משותף, והם רצים בזמנים שונים ולא דטרמיניסטיים ועשויים גם דברים בחלק ה-remainder.

דיקטורה פירמל מה הופך פתרון ניהול הגישה למשאב לטוב וקבע חמישה תנאים:

1. מניעה הדדית - רק תהליך אחד מריץ חלק קרייטי בכל פעם.
2. התקדמות - אם תהליכיים רוצים להיכנס לחלק הקרייטי, מישחו יכנס לחלק הקרייטי.
3. היעדר הרעבה - אף תהליך לא יוכה אינסופית לכניסה לחלק הקרייטי.
4. כלויות - האלג' עובד על N תהליכיים עם הרבה מעבדים ולא מניח הנחות נוספות.
5. אין חסימה ב-remainder - אף תהליך שרצ מחוץ לחלק הקרייטי לא חוסם תהליכיים מלהיכנס לקטע הקרייטי.

פתרנו את הבעיה או בעורת חומרה עם פעולות אוטומיות כמו Test&Set, בעזרת ה-OS באמצעות סמאפור ומיאוטקס ובעזרת **אלגוריתמים** כמו אלג' המאפייה והאלג' של פיטסרון.

דוגמה הקוד הבא הוא פתרון לבעית קטע הקרייטי. הוא מקיים את כל התנאים של דיקטורה חז' מהרעבה כי יכול להיות שתמיד תהליך יפספס את הבדיקה ותהליך אחר יבודוק ויידחף לפניו, لكن זה לא פתרון חוקי.

```

int lock=0; //shared variable
.....
while (TestAndSet(&lock))
    \\busy wait
critical section
lock = 0

```

איור 83 : קוד למניעה הדזית

דוגמה באמצעות סמאפור, שהוא מספר שימושים ומורידים באופן אוטומי, אפשר לפטור את בעיית קטע הקוד הקרייטי.

- **זמןן**: צריך לקבוע מי יירץ על המעבד (ים). יש הרבה אלג' לכך, ביניהם FCFS (הראשון שבא), SJF (המשימה הכי קצרה), SRTF (כל פעם שבא חדש נבדוק למי נשאר היכי פחות זמן), זמןן קדיםיות ו-RR (שראינו שבאופן כללי פותר את הבעיה היכי טוב). אלג' זמןן נמדד לפי כמה זמן המעבד מರיץ תהליכיים ולא מחשב זמןן (כמה שיותר), זמן המתנה (כמה שפחות) זמן turnaround, כלומר כמה זמן לוקח משמשימה מגיעה ועד שהיא מסיימת (כמה שפחות).

- **ניהול זיכרון**: הזיכרון מורכב מרגיסטרים עם מעבד, מטמון שמקצר את הדרך לזכרון, הזיכרון המרכזי, והתקני I/O לרבות דיסק. ההבדלו בין כתובות וירטואליות (רציף וגודל מאד, וירטואליות לתחלת) וכ כתובות פיזיות (מה שיש בזכרון, שהוא די קטן). ה-MMU ממיר מירטואלי פיזי בכל גישה לזכרון.

את המיפוי אפשר לעשות בכמה דרכים - בהקצאה רציפה שהיא לא פרקטית, סגמנטציה שגרמה לפרגמנטציה חיצונית ודף שבו משתמשים.

בדף חילקו את מרחב הכתובות הווירטואליות לדפים ואת הזכרון למסגורות כאשר גודל דף שווה לגודל מסגרת. לא כל הדפים מופיעים במסגורות. כsharp; דף שלא מופיע, משתמשים בזיכרון אלג' לירוש מסגורות, שהאופטימלי הוא שלbialdi ובמקרים מסוימים ב-Second Chance Clock Algorithm.

את טבלת הדפים שמחזיקה את המיפוי גם צריך לשנות לפחות פעמיים - טבלה שטוחה דורשת הרבה מקום ולכך היו גם כמה פתרונות:

1. טבלת דפים הפוכה - מיפויים מסגורות לדפים ואז כדי לבדוק אם דף נמצא צריך לעבור על כל המסגורות.
2. טבלת דפים היררכית - טבלת הדפים הראשית מצביעה לעוד כמה עמוקים של טבלאות דפים שבעמוק התהנתן יש מיפוי לדפי תוכן.
3. טבלת האש לדפים - תופסת דף מעט זיכרון ומציאת דף היא מהירה.

כל שיש יותר עמוקים בטבלה, זה דורש יותר גישות לזכרון (שזה יקר). כבר בטבלה שטוחה זה דורש קריאה אחת לטבלה ואחת לכתובות עצמה.

כדי לקצר תהליכיים, יש מטמון גם למיפוי, ה-TLB שמאיצ את התרגומים ומנע גישה חוזרת כל הזמן לטבלת הדפים כשלא באמת צריך.

- **תקשורת**: מערכת הפעלה אחראית על תקשורת בין תהליכיים בכמה דרכים שונות:

- זיכרונו משותף (ואז צריך לנחל מניעה הדדית).
 - פיעפים - עורך תקשורת בין שני תהליכים. הציגו מרכיב משנה קבצים (לא באמצעות קבצים אבל אבסטרקציה של ה-OS) שככל אחד מהם הוא המקום שבו תהליך אחד כותב לאחר.
 - סוקטים (באמצעות TCP בשבייל תקשורת אמינה ו-UDP לתקשורת מהירה).
- ניהול תהליכים : הפ' `fork` יוצרת תהליך חדש (בן) שיוצרת עותק זהה של תהליכי האב (עד כדי `pid`) ושניהם ממשיכים לרווח מאוותה הנקודה. נגרום להם לעשות דברים שונים ע"י בדיקת ה-`pid` שלהם.
- `dup` היא פ' שימושתית FD, כולומר למעשה `open` לאותו הקובץ שוב (ה-`offset` באותו המיקום) ו-`dup2` שימושתית FD אל תוך FD קיימים.

דוגמה השתמש בפיעפים בשבייל תקשורת בין תהליכי לבן שלו. ערך ההחזרה של `fork` הוא 0 אם "סאנחו בתוך הבן".

```
#include <unistd.h>
#include <stdlib.h>
#include <sys/wait.h>
int main() {
    int pipefd2[2];
    pipe(pipefd2);
    if (fork() == 0) {
        dup2(pipefd2[1], STDOUT_FILENO);
        close(pipefd2[0]);
        close(pipefd2[1]);
        execl("/bin/ls", "ls", NULL);
        exit(EXIT_FAILURE);
    }
    if (fork() == 0) {
        dup2(pipefd2[0], STDIN_FILENO);
        close(pipefd2[0]);
        close(pipefd2[1]);
        execl("/usr/bin/file", "file", "-f-", NULL);
        exit(EXIT_FAILURE);
    }
    close(pipefd2[0]);
    close(pipefd2[1]);
    wait(NULL);
    wait(NULL);
    return 0;
}
```

איור 84 : קוד לתקשורת בין תהליכים עם פיעפים

הקוד ייצור בן שידروس את `stdout`, סוגר את הקבצים שפתחו בשביילו וקורא ל-`ls` שתוצאתה תיקנס לקובץ הראשון של הפיפ. הבן השני שנוצר דורס את `stdin` כך שהוא מקבל מיידע דרכו. הבן הראשון שולח רשימת קבצים באמצעות `ls` והבן השני מקבל

את המידע הזה ורץ file.

Iseek מזיהה את היחסט בתוך הקובץ ביחס לסוף הקובץ, תחילת הקובץ, או המיקום הנוכחי. כדי לדעת איפה היחסט כרגע, נוכל לקרוא לפ' עס 0 ביחס להיחסט הנוכחי ונקבל את היחסט החדש (שהוא הישן).

- **מערכת קבצים** : קריאה מהדיסק זה יקרה ולכון מערכת הקבצים מנסה לסייע את מספר הגישות, בין היתר באמצעות ה-buffer cache. בינויקס משתמשים ב-node-i שמכיל מטא-דאטה על הקובץ לרבות גודלו ומתי עודכן לאחרונה, וגם מצביעים לבולוקים שבהם הוא נמצא בזיכרון (עם מצביעים ישירים, עקיפים, שניוניים, ושלישיים).
טבלת הקבצים מתחולקת לשולש:
 1. טבלת ה-FD של כל תהליך בנפרד - כל קובץ מופיע פעם אחת לכל פעע שנפתח והכניסה מחזיקה מצביע לטבלת הקבצים הפתוחים.
 2. טבלת הקבצים הפתוחים - טבלה כללית לכל ה-OS שמחזיקה כניסה אחת לכל פעע שקובץ נפתח והיחסט שבו התחלך נמצא בקובץ.
 3. טבלת ה-node-iים - טבלה שבה כל ה-node-iים של קבצים פתוחים, כניסה אחת לכל קובץ.

הוא קובץ עם מטא-דאטה שלא מכיל את שם הקובץ שלו והוא מתייחס או ה-path אליו, והקובץ מחזיק כמה ופרנסים יש אליו וריך כשיש 0 הוא נמוך.

יש פ' שנראית link שמקבלת נתיב ישן חדש ושם בנתיב החדש hardlink לנתיב הישן.
unlink מסירה את הلينק לקובץ (מקטינה את מספר הזריםים).
softlink הוא מצביע ישיר לנתיב של הקובץ ואם הקובץ המקורי נמחק, הקישור לא יעבד.

- **וירטוואלייזציה** : VM-ים עושים אבטורקציה לחומרה ומעלה, קונטינרים עושים אבטורקציה רק ל-OS תחת אותה חומרה (באמצעות כמה סוגים שונים של HV).

בליוקס יש קונטינרים שנקראים LXC Namespaces השונים מהניטיב ב-NS שונות (כמו pids חדשים, מערכת קבצים חדשה וכו').
אפשר לבקר (מלשונו בקרה) את ה-NS באמצעות cgroups שימושיים באבטורקציה כקבצים.

שאלות ותשובות

- כיצד ניתן למשתמש מותמך תהליכי ברמת המשמש?
- להשתמש בכלל' שעוצר תהליך (כלומר freempt) באמצעות שליחת SIGSTOP והמשכת ריצת תהליך באמצעות SIGCONT).
- במערכת הקבצים המסורתית המבוססת node-i, כמה גישות לדיסק נדרשות לכל הפחות כדי לקרוא את הבית 8097-8 בקובץ /home/moishe/mail (אם כל בלוק הוא בגודל 2^{10} , יש 6 מצביעים ישירים ואחד מכל עקייף, וכל מדרייך מאוכסן בבלוק יחיד)?
כדי להגיע לקובץ צריך 3.1-node-iים ומדרייכים (מדרייך השורש, node-i ומדרייך, node-i ומדרייך, אחד ל-single-indirect ואחד לתוכן הקובץ).

- נתון הקטע הבא לביעות הקטע הクリיטי,

ד. להלן פתרון לביעית הקטע הクリיטי לשני תהליכיים (0 שקול ל FALSE ו-1 שקול ל TRUE):

```
At process i ∈ {0,1}
shared boolean flag[2] = {false};
shared int turn = 0;
do
{
    turn = 1-i;
    flag[i] = false;
    while !(flag[1-i] && turn==1-i);
    critical section
    flag[i] = true;
    remainder section
} while(true);
```

איור 85 : קוד לפתרון בעיית האזור הクリיטי

- האם המימוש עשוי לגרום לדד-לוק? כן, אם קורה קונטקט סוויז' בדיק אחורי השמת false בדגל של התהליך עצמו, שני הדגלים הם false ואז שניהם יכנסו ל-while ולא יצאו ממנה אף פעם.
- האם יתכן זדLOCK אם נוריד את השורה `flag[i]=false` ונתחל `{flag[1]={true};` לא, כי אם קורה קונטקט סוויז' בדיק לפני ה-turn משנהו ותנאי ה-while לא מתקיים כי שני התנאים בפנים הם true ואז מישחו יכנס.
- נניח שמבצעים את השינויים בסעיף הנ"ל האם מתקיים תוכנה מניעה הדדיות? לא, נניח שתהליך 0 מגיע, משנה את turn ל-1 וכן לקטע הクリיטי, ואז יש סוויז' ותהליך 1 משנה את הפרמטרים בהתאם ובמס לקטע הクリיטי כי תנאי ה-while לא מתקיים.
- מהו הבדל אחד בין תהליך לבין חוט ברמת הקרןלהו (יש 4 אפשרויות שלא אציג)? תהליכי זוקרים לティווך של מערכת הפעלה כדי לתקשר וחוטים ברמת הקרןלה לא זוקרים (כי הם יכולים לעשות זאת באמצעות זיכרונו משותף).

שבוע VII | חזרה

תרגול

שאלה ענו את השאלה הבאה

```

Char* buffer = new char[1];
Int fd = open ("some-file");
//...
read (fd, &buffer, 1)

```

(6 נק'). הניחו כי תהליך מבצע את הקוד הבא:

כלומר, הקוד מבצע פקודת קרייה מהדיסק של בית בודד על ידי שימוש בפונקציה `read`. בנוסף הניחו כי המשתנה `buffer` נמצא בדף (page) שאינו מופיע למסגרת (frame) חוקית בזיכרון (כלומר, ה-bit `valid` אינו דלוק).

הניחו גם כי כל ה-`system calls` מסתיימות בהצלחה וערך ההחזרה של `read` הוא 1.

נעסק בשורה الأخيرة בקוד בלבד, שמבצעת פקודת `read`. ציינו, לפי הסדר, אילו פסיקות (interrupts and traps) יהיו במערכת? עבור כל פסיקה ציינו את הסוג שלה, והסבירו בקצרה (עד 2 שורות) מדוע פסיקה זאת מתרחשת ואיזה רכיב במערכת מבצע אותה.

הקוד פותח קובץ וקורא ממנו אל תוך אפר. אם `read` החזיר 1 זה אומר שהוא קרא בית אחד (כרצוי).

1. ראשית יהיה trap כדי לבצע את ה-`syscall` המבוקש.

2. כשהדיסק יסיים יהיה אינטראפט חומרה במסגרת סכמת ה-DMA.

3. page fault כי אנחנו רוצים לכתוב לדף שלא מופיע (אינטראפט פנימי).

4. כשהדיסק מסיים להביא את הדף, יהיה עוד אינטראפט חומרה.

שאלה

• נניח כי כעת רץ במערכת חוט מס' 3 של תהליך a וב-MMU מסוימת דף X למסגרת Y. האם המיפוי ישמר אם המערכת תריץ

חותם אחר של אותו תהליך?

כן כי הזיכרון משותף בין חוטים.

• האם המיפוי ישמר אם המערכת תריץ חוט של תהליך אחר?

לא כי לכל תהליך יש טבלת דפים נפרדת.

שאלה

קראו את הנתונים הבאים

שונה פילוסופים יושבים סביב שולחן עגול במסעדת. במרכז השולחן מונחים ארבעה מזלגות וארבעה סכינים. הפילוסופים מאוד רעבים, אך אוכלים במנות קטנות. לכן, הם רוצים לקחת סכין ומצלג, לאכול קצת, ואז להחזיר את הסכין והמצלג ולונוח. מכיוון שהם רעבים, הם חוזרים על כך שוב ושוב.

הפילוסופים ממוספרים בין 0 ל 7, וכל פילוסוף משנתה i פרטיו בו נמצאו מספרו הסידורי. הם משתמשים בסמפורים knives, forks שמאוחתלים כל אחד לערך 4. נניח כי מערכת הפעולה משתמשת את הסמפורים הנ"ל וכן כי הסמפור הוגן: החסימה והשחרור של תהליכי מבצע FIFO (אם תהליך i נחסם על הסמפור לפני תהליך j אז תהליך i ישוחרר מהסמפור לפני תהליך j).

לכל אחד מהפתרונות הבאים צין האם יתכו קיפאון deadlock או הרעה starvation. נמקו את תשובה שלכם.

.1

```
while(true)
{
    down(forks)
    down(knives)

    //eat

    up(knives)
    up(forks)

    //rest
}
```

Code for philosopher i

לא יהיה דד-לוק כי נעילת המשאבים נעשית בסדר זהה (הפוך ביציאה) כך שלא ייתכן שנחזיק את knives אבל לא את forks וכו'.

לא תהיה הרעה בגלל שני הסמפורים שקובעים הלכה למעשה מי אוכל הם הוגנים.

.2

```

while(true)
{
    If(i<4)
    {
        down(forks)
        down(knives)
    }
    else
    {
        down(knives)
        down(forks)
    }

    //eat

    up(knives)
    up(forks)

    //rest
}

```

Code for philosopher i

יהיה דד-лок כי יכול לקרות ש- $i = 0, 1, 2, 3$ ירימו את forks וכולם בדיק יחתפו סוויז' לפני שיספיקו לחת את knives, ואז

ירימו את knives ולא את forks ואז אף אחד לא יוכל להרים שום דבר.

יש הרעה כי דד-лок הוא מקרה פרטי של הרעה.

3. אותו הדבר רק עם $i < 3$.

לא יהיה דד-лок כי רק שלושה יכולים לנעול את forks קודם וזו תמיד אחד מהחמים האחרים יצליח ל特派 גם את knives וגם את forks.

בגל שהסמאפורים הוגנים, לא תהיה הרעה (אין באמת סיבה שתהייה כי גם אם סדר ההרמה הפוך הם כיחידה אחת הוגנים).

שאלה קראו את הנתונים הבאים

מערכת הפעלה מסוימת משתמשת במערכת קבצים מסוג e-node-i-

גודל בלוק במערכת היו 4KB

גודל מילה היו 8 בתים, שזה גם גודל מצביע לבlok

מבנה עץ ה-e-node-i הוא:

- מביצעים ישרים 64
- הצביעות עקיפות חדירות – single indirect pointers
- הצביעות עקיפות כפולות – double indirect pointers

המערכת משתמשת ב-Buffer Cache בגודל 257 בלוקים בלבד, והחלפת בלוקים בו מבוצעת באמצעות שיטת LRU.

שימוש לב: ה-e-node-inode נשמר באוצר שונה של הזיכרון כדי לאפשר גישה לקובץ גם אם הוא לא היה בשימוש לאחרונה.

הערה: ה-block Cache נקרא לפעמים גם Disk Cache או (בחומר דיקן), וגם Buffer cache. מדובר באותו cache.

1. מה גודל הקובץ המקסימלי שנitinן לייצג במערכת קבצים זו?

כל בלוק נתון לנו $4KB = 2^9$, 64 המכבייעים הישירים מתאימים לבлок אחד בלבד. כל מכבייע עקייף ייחיד נתון לנו $512 = 2^{9+2} = 2^{11}$ בלוקים נוספים ועקייף כפול נתון לנו $2^{18} = 2^{11+7} = 2^{28}$ כלומר סה"כ

$$64 \cdot 2^{12} + 4 \cdot 2^9 \cdot 2^{12} + 4 \cdot 2^{18} \cdot 2^{12} = 2^{15} + 2^{23} + 2^{32}$$

2. תחילה קורא קובץ בגודל 8MB מהdisk, באופן רציף, בלוק אחר בלוק, מתחילה ועד סופו. כמה גישות תבצע המערכת לדיסק לקריאת בלוקים? הניחו שהתחילה יודע היקן `node-i` של הקובץ אבל עוד לא ניגש אליו, שה-`buffer cache` ריק, שמתבצעת קריאה ל-`node-i` לפני כל קריאה לבlok נתונים והתחילה זהה הוא היחיד שמשפיע על ה-`buffer cache`.
נצרך לחשב כמה בלוקים תופס הקובץ. $2048 = 2^{11} = \frac{2^{23}}{2^{12}}$ בלוקים, כלומר משתמש ב-64 יישירים, וארבעת העקיפים היחידים לא בccoli אבל מה שחשוב זה שאנו לא זולגים לעקייף כפול).
בשימוש בעקיפים היחידים, נכנס את בלוק המכבייעים לזכרו ולא נזיהר אותו עד תום השימוש כי אנחנו משתמשים ב-LRU. הגישה הראשונה היא `node-i` והוא גם נשאר במאפר-קאש כל הזמן, 64 הבאות לבlokים הישירים, אז עוד $4 + (2048 - 64) = 2048$ כי אנחנו צריכים לקרוא את הבלוקים העקיפים אבל רק פעם אחת.

כלומר סה"כ

$$1 + 64 + (2048 - 64) + 4 = 2^{11} + 4 + 1$$

3. איך הייתה משתנה התשובה לסעיף הנ"ל אם לא היה `buffer cache`?
הינו צריכים לקרוא כל פעם מחדש את הבלוקים העקיפים. כלומר היה לנו את החישוב הנ"ל רק עם $(2048 - 64) \cdot 2$ במקומות סתם $(2048 - 64)$, כאשר כל השאר ישאר זהה כי ה-`node-i` הוא בזיכרון.
4. איך הייתה משתנה התשובה לסעיף 2 אם היה `buffer cache` אבל ניהול שלו היה ב-FIFO?

עד העקיפים הכל ישאר אותו הדבר. על כל עקייף נקרא אותו ואז $256 - 1 = 257$ בלוקים של הקובץ עד שנצרך להכניס אותו שוב לזכרו, כלומר נקבל את החישוב מהסעיף הראשוני ועוד 8.

שאלה נתונה מערכת עם מזמן תור קדימיות פידבק רב שלבי עם ארבעה שלבים (1 בעדיפות הכי גבוהה ו-4 בעדיפות הכי נמוכה). קראו את

הנתונים הבאים

כשההליכים עוברים במצב ready הם נכנסים לתור מס' 1, המומッシュ round robin עם $q=20\text{ms}$. תהליך שלא סיימ (או לא עבר למצב waiting) עד סוף ה-quantum (waiting) שלו עבר לתור מס' 2 המומッシュ round robin עם $q=30\text{ms}$. תהליך שלא סיימ (או לא עבר למצב waiting) עד סוף ה-quantum (waiting) שלו עבר לתור מס' 3 המומッシュ round robin עם $q=40\text{ms}$. תהליך שלא סיימ (או לא עבר למצב waiting) עד סוף ה-quantum שלו עבר לתור מס' 4 המומッシュ first come first served (waiting).

זמן ביצוע context switch במערכת הוא 10ms.

בנסיבות א'-ג' נניח כי ה-scheduler מתזמנת תהליכיים בין התורדים strict priority: תהליכיים בתור בעל עדיפות גבוהה יותר מזו של תהליכיים בתור בעל עדיפות נמוכה יותר.

1. נניח כי שלושה תהליכיים עוברים במצב מוקן בו זמניות וזמן העבודה של כל אחד מהם הוא 150ms . חשבו את ה-turnaround time המומוצע (זמן שלוקח מה כניסה למתזמן ועד סיום הריצה).

שלושת תהליכיים ירוצו 20ms כאשר לפניו יהיה קוונטקס סוויצ'ינג של 10ms (חוץ מבריצה הראשונה שבה אנחנו מניחים שהתליק כבר טען לזרקתו).

שלושתם ירוצו עוד 30ms עם 3 קוונטקסים סוויצ'ינג לפני הריצה.

שלושתם ירוצו עוד 40ms עם 3 קוונטקסים סוויצ'ינג. עד לשלב זה עברו 360ms .

שלושתם ירוצו עד הסוף (נותרו 60ms עד תום הריצה) עם 3 קוונטקסים סוויצ'ינג, אשר הראשון יסיים לאחר 60ms נוספת, 560ms , כלומר 420ms , השני יסיים אחרי $60+10\text{ms}=70\text{ms}$ וכלומר 490ms והאחרון יסיים לאחר 490ms .

2. נניח כעת כי שלושה תהליכיים מבצעים לולאה נוספת בתחום ב- 30ms page fault של 10ms שנגמר ב- 30ms שלוקח 30ms לאחריו הוא חוזר על החישוב וחזור חלילה. בנוסף, קיימים תהליכי רביעי המבצע חישוב שזמן העבודה שלו הוא 150ms (לא לולאה). חשבו מהו הזמן waiting time של התליק הרביעי.

השלושה הראשונים ירוצו 10ms עם קוונטקס סוויצ'ינג של 10ms . הרביעי ירוץ ל- 20ms וירד לתור השני. עד השלב הזה מספיקים כל תהליכי האחרים לחזור (כולם מספיקים לחזור והתליק השלישי בדיקת מס' 30ms של ההמתנה שלו).

אותו הדבר קורה שוב רק שהפעם התליק הרביעי לא ירוץ כי עד כל תהליכי מספיקים לחזור כך שתור 1 תמיד מכיל תהליכיים כלומר תליק 4 מורעב וירוץ לנצח. זמן ההמתנה שלו הוא אין סוף.

3. עתה נשנה את המזמן מעט: נעלם את העדיפות של תליק ב-1 אם תוזמנו 100 תהליכיים (לא בהכרח שונים) מהעדיפות שמעליו בלי שהוא תזמין. האם השינוי הזה יוביל לשינוי בזמן ההמתנה של תהליכיים מהסעיף הנ"ל (מעבר תהליכי אינסופיים נסטכל על זמן ההמתנה בשעה הראשונה)? אין צורך בחישובים מדויקים.

כן, יהיה שינוי. שלושת תהליכיים יקבלו זמן תגובה יותר גבוהה (כי התליק הרביעי בשלב כלשהו ירוץ יותר מאשר בלי השינוי), כלומר הוא תפס קצר מזמן הריצה שלהם) ולתקליק הרביעי יהיה זמן תגובה סופי במקום אינסופי, כלומר ירידת.

4. האם המערכת מהסעיף הנ"ל מקיימת היעדר הרעה?

כו, כי מעלים עדיפות רק עם תזומנו 100 תהליכי בעדיפות מעל, לא בכל העדיפות מעל. לכן יכול להיות שתהליך ירד לעדיפות 3 ואז ירצו אינסופית תהליכי בפרטים קצרים בעדיפות 1, כך שאף פעם לא יתקיים התנאי שבו נعلاה את הדיפות של התליש המורעב.

שאלה קראו את הנתונים הבאים

נתון מחשב עם מבנה הזיכרון הבא:

- זיכרון לוגי לכל תהליך 2^{32} בתים (bytes)
- זיכרון פיזי בגודל 2^{24} בתים.
- גודל כל דף/מספרת הוא $2^{12} = 4KB$ בתים.
- טבלאות הדפים ממושאות כפי שנלמד בכיתה (במימוש הבסיסי), ללא אלגוריתמים נוספים
- (למשל hashing) שנועדו לשפר את ביצועיהם.

1. מהו גודל טבלת הדפים (בדפים) אם החלטנו למש את טבלת הדפים כטבלת דפים הפוכה, כאשר רשומה בטבלה היא בגודל 4 בתים.

$$\text{יש } \frac{2^{24}}{2^{12}} \text{ מסגרות, כלומר סה'כ } 4 \cdot 2^{12} \text{ בתים לרשותם, שנכונות ב-4} = \frac{2^{12} \cdot 4}{2^{12}} \text{ דפים (החלק היא בגודל של כל דף).}$$

2. מהו זמן התרגום הממוצע של ה-MMU בהנחה שקריאה של 4 בתים רצופים לוקחת 2^{10} ושההסת' לכתובות שכן ממופה לזכרון הפיזי היא 0.5. ניתן להניח כי הטבלה נמצאת כולה בזיכרון הפיזי. אם התרגום הסטיים ב-page fault או שהוא page fault במהלךו, יש להתחשב רק בזמן עד ה-page fault (כלומר מעוניין אותנו רק החישוב האם יש page fault או לא). נctrיך לעבר שורה שורה, כך שאם הוא ממופה, בממוצע הוא יושב באמצעות הטבלה (תווחלת קלאסית) ואם הוא לא ממופה נגיע עד לסוף הריצה. כלומר נידרש ל-

$$\frac{1}{2} \left(\frac{1}{2} \cdot 2^{12} \right) + \frac{1}{2} (2^{12}) = 2^{10} + 2^{12} = 1024 + 2048 = 3072$$

מילוי-שניות בממוצע.

3. מהו גודל טבלת הדפים (בדפים) אם החלטנו להשתמש בטבלת דפים רגילה - לא הפוכה ולא היררכית - כאשר כל רשומה תופסת 4 בתים.

$$\text{יש לנו } 2^{20} = \frac{2^{20} \cdot 4}{2^{12}} = \frac{2^{22}}{2^{12}} = 2^{10} \text{ בתים לרשותם, שהם } \frac{2^{32}}{2^{12}} \text{ דפים.}$$