OS - תקציר

15 באוגוסט 2012

איני לוקחת אחריות על מה שכתוב כאן, so tread lightly
אין המרצה או המתרגלים קשור לסיכום זה בשום דרך.
הערות יתקבלו בברכה - noga.rotman@gmail.com.

1 סיגנלים, אינטרפטים, אקספשנים ו־$traps$

1.1 הגדרות בסיסיות

1.1.1 מבנה הזיכרון

- הזיכרון הראשי - ממוקם על צ'יפים בתוך המחשב (מחוץ למעבד). ההוראות של תוכנית והמיעד של תהליכים נשמרים בזיכרון זה.
- זיכרון חיצוני - דיסק. על המידע הנשמר על הדיסק לא נמחק כאשר המחשב מכובה. זהו זיכרון משני.

בזיכרוון הראשי יש פחות מקום מהדיסק הקשיח. הדיסק הקשיח יכול לכתוב ולקרוא מידע מתוך ואל הזיכרון הראשי. מהירות הגישה של הזיכרון הראשי מהירה בהרבה מזו של הדיסק הקשיח.
תוכניות נשמרות על הדיסק עד שהן נטענות לזיכרון, ואז הן משתמשות בדיסק כמקור וכיעד המידע.

1.1.2 קרנל מול פרוסס

הגדרה 1.1 הקרנל - הקוד של מערכת ההפעלה. נחשב בבחינתנו קוד שהוא trusted - טוב, ונכתב ע"י "גורמים טובים".

הגדרה 1.2 process - אינסטנס של תוכנית (בהגדרה, כל מה שרץ ביוזר מוד הוא פרוסס). כל הזמן יש process אחד ויחיד שהוא פעיל ב־CPU, השאר מחכים ל־CPU להתפנות. קוד זה אינו trusted, ויכול להיות שיש בו באגים.

ה־CPU יכול לרוץ בשני מודים:

- user mode - מוד בו פרוססים לא יכולים לגשת לחלקים בזיכרון שהוקצעו לקרנל או לתוכניות אחרות.
- כאשר פרוסס ב־user mode רוצה להשתמש בשירות המסופק ע"י הקרנל, על המערכת להחליף זמנית ל־kernal mode - לאחר שהפעולה המבוקשת מתבצעת, המעבד חוזר ל־user mode. קוד שרץ במוד זה יכול לגשת למשאבים ראשיים.
כל הקרנל, שאינו תהליך אלא שולט בתהליכים, מתבצע אך ורק ב־kernal mode. לאחר סיום הפעולה, המעבד חוזר ליוזר מוד.

1.2 אינטרפטים

הגדרה 1.3 אינטרפט הוא סיגנל למערכת ההפעלה המציין כי אירוע התרחש, ובעקבותיו מתבצע שינוי ברצף ההוראות שמבוצעות ע"י ה־CPU. ישנם שני סוגים של אינטרפטים:

- אינטרפט של התוכנה (software interrupts) - כוללים אקספשנים ו־traps.
- חיצוני למעבד שמקורו בחומרה (hardware interrupts). במקרה זה הסיגנל נוצר ע"י device כגון המקלדת, העכבר או system clock (המשתמשים בו על מנת לתאם את פעולות המחשב). אינטרפטים אלו הם אסינכרוניים, שכן הם נוצרים מחוץ לתוכנית.

לרב, מערכת ההפעלה לא צריכה להתעלם מאף אינטרפט.

טיפול באינטרפט דומה לטיפול בקריאה לפונקציה - החומרה קוראת לפונקציה (ההנדלר) לטפל בו.
המנגנון הבסיסי של האינטרפטים הוא:

- ה־CPU מקבל את האינטרפט, שומר את הרגיסטרים החיוניים, ומחליף אותם בערכים המתאימים להנדלר.
- שליטה עוברת למערכת ההפעלה.
- המצב הנוכחי נשמר.
- הבקשה מטופלת.
- המצב הקודם משוחזר.
- השליטה חוזרת לפרוסס.

1.2.1 אקספשנים

הגדרה 1.4 סוג של software interrupt. אקספשנים לא נגרמים ע"י מקור חיצוני, אלא כחלק מריצה רגילה של תוכנית, מתוך המעבד עצמו. הם נוצרים כאשר משהו מתרחש הגורם לכך שהמעבד לא יכול לטפל בהוראה - בין אם קריטית (חלוקה ב־0, segmentation fault - זיכרון שעדיין לא הוקצה), שלרב גורמת לתוכנית להסגר, או זמנית.

יש שני סוגים של אקספשנים:

- שגיאות - למשל, נניח שיש תוכנית שמנסה לחלק באפס. המעבד לא יודע מה להחזיר במקרה זה, ונגרם אקספשן. אזי, המערכת תהרוג את הפרוסס. עוד דוגמא - פנייה לכתובת שגויה. לאחר אקספשן של שגיאה, על המערכת להמשיך להוראה הבאה בתור, שכן אם היא תחזור על ההוראה שגרמה לשגיאה, ניכנס ללופ אינסופי.
- לא משגיאה, אלא משהו שהמערכת צריכה לטפל בו. למשל, פניה למידע שלא נמצא כרגע בזיכרון הפיזי. האקספשן הזה "מעיר" את מערכת ההפעלה, שמעבירה את המידע לזיכרון, מבלי שהמשתמש בכלל שם לב לכך. במקרה זה, מערכת ההפעלה תגרום להפעלה חוזרת של הפעולה (שלא יכלה להתקיים קודם לטיפול), כך שהמשתמש לא ישים לב שקרה משהו נוסף.

1.2.2 trap

הגדרה 1.5 סוג נוסף של software interrupt. מתרחשת בעת ריצה רגילה של התוכנית, אולם בניגוד לאקספשן, היא לא תוצאה של טעות. היא הוראה המבקשת ממערכת הפעלה לעשות משהו - היא קוראת ל־system call (פתיחת קובץ, סגירת קובץ, לבדוק הזזת עכבר, תקשור עם פרוסס אחר... כל דבר שאינו בזיכרון של הפרוסס). למשל $fopen$ היא מעטפת ל־system call - צורך לקריאה של מידע מקובץ.. פקודה זו מעבירה את המעבד לקרנל מוד, וקוראת ל־trap handler לטפל בפקודה הנדרש. ישנה טבלה של כל ה־system calls המותרים.
לאחר ביצוע הטראפ, מבצעים את הפעולה הבאה בתור.

1.3 system call

הגדרה 1.6 system call הוא מנגנון בו משתמשת תוכנית על מנת לבקש שירות ממערכת ההפעלה. היא מספקת את הממשק בין תהליך לבין מערכת ההפעלה עצמה (fopen למשל).
system call יכולה להכשל מכל סיבה כלשהיא - יש לבדוק תמיד ערך החזרה, ואם נכשלה לבדוק מדוע בעזרת קוד השגיאה.

1.4 סיגנלים

הגדרה 1.7 סיגנל הוא דרך של מערכת ההפעלה לשלוח הודעה לפרוסס על אירועים "חשובים" שונים. סיגנלים גורמים לפרוסס לעצור את מה שהוא עושה באותו הרגע, ולטפל בהודעה באופן מיידי. הפרוסס מפעיל קוד ספציפי כדי לטפל בסיגנל המסויים. הפרוסס יכול גם לקבוע מה לעשות כשהסיגנל יגיע אליו - מלבד סיגנלים מסויימים.

סיגנלים נוצרים על ידי מערכת ההפעלה, ומתקבלים ומטופלים ע"י פרוסס. אינטרפטים לעומת זאת נוצרים על ידי החומרה, ומתקבלים ומטופלים ע"י מערכת ההפעלה.
סיגנלים יכולים להגרם ע"י:

- קלט אסינכרוני מהמשתמש - למשל ^C ($SIGINT$).
- המערכת או פרוסס אחר, למשל אם הזמן שהוקצב לפרוסס אזל ($SIGALRM$).
- אקספשנים בחומרה הנגרמים ע"י הפרוסס, כגון גישה לא חוקית לזיכרון ($SIGSEGV$).

- דיבגר המבקש לעצור או להקפיא את ריצת הפרוסס.

אפשר לשלוח סיגנלים ע"י המקלדת ומה־*shell*.
הקרנל מטפל בסיגנלים תוך כדי ריצת הפרוסס, כלומר הפרוסס צריך לרוץ על מנת לטפל בסיגנלים. ישנם שלושה סוגים של טיפולים בסיגנלים:

- הפרוסס מבצע exit (דיפולטיבי לחלק מהסיגנלים).
- הפרוסס לא עושה כלום - או, עושה ignore (דיפולטיבי לחלק מהסיגנלים).
- הרצה של סיגנל הנדלר - פונקציה שנמצאת בקוד של הפרוסס. צריך להגיד למערכת ההפעלה להפעיל את הפונקציה המבוקשת. ההנדלר כמובן רץ ביוזר מוד (שכן הוא רץ כחלק מהפרוסס).

כאשר הקרנל יוצר פרוסס חדש, נוצרת לו טבלה שאומרת לו מה לעשות במקרה של סיגנלים - יש לה מצב דיפולטיבי, אבל אפשר לשנות אותה (אבל לא לכל הוראה - נראית זו בהמשך). להודיע על סיגנל משמע לבצע את הפונקציה שמתאימה לסיגנל בטבלה - מערכת ההפעלה פשוט מעדכנת את ה־program counter למקום של הפונקציה המבוקשת (כמובן מתבצעת שמירה של המצב הנוכחי).
"זה לא יהיה רעיון טוב" לאפשר לשנות את הפעולה של כל הסיגנלים. ישנם סיגנלים שאת פעולתם לא ניתן לשנות, למשל kill, stop. לכל סיגנל יש דיפולט משלו.
ישנן שתי פונקציות לטיפול בסיגנלים הקבועות מראש בהן ניתן להשתמש (בפונקציה סיגנל ובפונקציות אחרות):

- SIG_IGN - אומר להתעלם מהסיגנל. אפשר לשמש באופן כללי בפונקציה ריקה (זה לא בדיוק אותו דבר).
- SIG_DEF - גורם לסיגנל לחזור לערכו המקורי בטבלה.

כיוון שהסיגנלים יכולים לקרות בכל רגע נתון (אסינכרוניים), הם יכולים לקרות בזמן שסיגנל הנדלר אחר רץ, בזמן שאנחנו לא רוצים שיפריעו לנו (בתוך שורה, בתוך פעולה שתתבצע שוב מההתחלה אם נריץ את הפעולה של הסיגנל), וכו'. מצב שכזה נקרא races. לכן נרצה למנוע מהסיגנל הנדלר לרוץ בזמנים בעייתיים. הדרך להתמודד - שמים חסמים (block and unblock) במקומות הרצויים - הסיגנל הנדלר לא ירוץ עד שחרור החסימה. נרצה לחסום כמה שפחות קוד.
בשביל זה משתמשים בפונקציה:

$$\text{int sigaction}(int\ sig, struct\ sigaction * new\_action, struct\ sigaction * old\_action)$$

action - סיגנל הנדלר+סיגנל מאסק+פלגס (לא נדבר על הפלגס).
פונקציה זו מגדירה את הפונקציה שיש להריץ כשיגיע סיגנל מסויים רק במהלך הקיום של הסיגנלר הנדלר הנוכחי. מגדיר את המאסק - המאסק באקשן לא מבקש לחסום דברים עכשיו, אלא כאשר הסיגנל הנדלר ירוץ. אפשר גם להשתמש בו כדי לראות מהו המאסק הנוכחי.

2 תהליכים ות'רדים

2.1 תהליכים

הגדרה 2.1 תהליך (פרוסס) הוא אינסטנס דינמי של אפליקציה - אבסטרקציה של "מחשב יחיד".

תוכנית היא זהות פסיבית, תהליך הוא אקטיבי. תוכנית נהפכת לתהליך כאשר קובץ executable נטען לזיכרון.

הגדרה 2.2 החלפת התהליך הרץ נקראת context switch - החלפת הקשר.

מצב של ה־CPU מורכב מ:

- Processor Status Word (PSW) - המוד, התוצאה של החישוב האלגברי האחרון (אפס, שלילי, overflow או $carry$), ורמת האינטרפט (אילו אינטרפטים מורשים ואילו יחסמו).
- Instruction Register (IR) - הכתובת של הפעולה שכרגע מתבצעת.
- Program Counter (PC) - הכתובת של הפעולה הבאה שתתבצע.
- Stack Pointer (SP) - הכתובת של ה־stack frame הנוכחי, כולל את המשתנים המקומיים של הפונקציה וה־return information.
- רגיסטרים נוספים.

מצב הזיכרון מורכב מארבעה סגמנטים:

- טקסט - הקוד של האפליקציה.
- מידע - מבני הנתונים של האפליקציה.
- Heap - אזור ממנו אפשר להקצות מקום דינמית בעת זמן ריצה.
- Stack - היכן שערכי הרגיסטרים נשמרים, משתנים מקומיים מוקצים, ומידע של function call נשמר.

שאר המצב של תהליך מורכב מ:

- PCB (Process Control Block):
  - מידע לחישוב הקדימות של התהליך ביחס לתהליכים אחרים.
  - מידע לגבי המשתמש המריץ את התהליך, בו משתמשים ע"מ לקבוע את הרשאות הגישה של התהליך.
- סביבה (רשום במספר טבלאות של מערכת ההפעלה):
  - פרמטרים של ה-GUI.
  - קבצים פתוחים בהם משתמשים לקלט ופלט.
  - ערוצי תקשורת עם תהליכים\מכונות אחרות.

המצבים האפשריים של תהליך הינם:

- new - התהליך נוצר.
- $running$ - ההוראות מבוצעות.
- $waiting$ - התהליך מחכה שאירוע כלשהוא יתרחש.
- $ready$ - התהליך מחכה להקצאה למעבד.
- $terminated$ - התהליך סיים את פעולתו.

2.2 ת'רדים

הגדרה 2.3 ת'רד הוא חלק מפרוסס, לפרוסס יכולים להיות הרבה ת'רדים.
מה שמאפיין אותו - יש לו קונטרול פלו משלו. כלומר, יש לו מיקום משלו בקוד, וזה מה שמבדיל אותו מת'רדים אחרים, והוא יכול לרוץ בנפרד מת'רדים אחרים. בשביל שלת'רד יהיה מקום מיוחד, יש לשמור נתונים מסויימים עליו:

- מיקום על הסטאק - פרמטרים לפונקציות, ועוד.
- רגיסטרים של הת'רד (מצב CPU) - למשל PC שהוא חלק מהרגיסטרים של ה-CPU.

שאר המשאבים של הפרוסס משותפים לכל הת'רדים יחדיו:

- קוד - כל ת'רד יכול לפנות לקוד.
- הזיכרון - אם הזיכרון משותף (משתנה גלובלי למשל), אז כל ת'רד יכול להשתמש בו כל אחד בזמנו, אם אחד מבזבז הרבה זיכרון, הוא מבזבז הרבה זיכרון לכל הת'רדים.
- קבצים פתוחים - אם ת'רד אחד פותח קובץ, אזי הקובץ פתוח עבור כל הת'רדים.
- ועוד...

תהליכים מול ת'רדים:

- כל ת'רד חייב לרוץ כחלק מתהליך מסויים.
- יתכנו מספר ת'רדים באותו תהליך - multithreading.
- תהליך - אוסף משאבים. ת'רד - ישות לזימון לריצה על המעבד.

למה משתמשים בת'רדים?

- ביצועים:
  - יצירת ת'רד מהירה פי 30־100 מיצירת תהליך.
  - ביצוע context switch מהיר פי 5.
  - כשת'רד אחד משתמש ב־CPU, אחרים יכולים לבצע blocking I/O שונים.
  - ניצול של כמה $CPUs$ במערכות SMP.
- תכנות מודולרי ־ ביצוע מטלות שונות במקביל עם מידע משותף:
  - $responsiveness$ לכל ת'רד.
  - שיתוף זיכרון בין ת'רדים לא מערב את הקרנל, בניגוד לסנכרון בין תהליכים.
- בלי ת'רדים קשה לבצע המתנה למספר ערוצי I/O (קלט מהמסך, הודעות מרשת תקשורת, מידע שביקשנו לקרוא מהדיסק...).
  אלטרנטיבה ־ תכנות בעזרת non-blocking primitives, למשל non-blocking read מהמקלדת ־ בשיטה זו חוזרים כאשר אין קלט, ומגלים מתי יש קלט בעזרת סיגנל או דגימה עם קריאת select(). הקושי בשיטה הזו היא בכתיבת התוכניות.
  תהליכונים לעומת זאת מאפשרים לכתוב תוכניות סדרתיות עם blocking calls ומצד שני מאפשרים מקביליות.

מבחינים בין שני סוגי מימוש עיקריים (יש מערכות הפעלה התומכות בשני סוגי המימוש):

- user-level threads ־ הקרנל לא משתתף בהפעלתם.
- kernal-level threads ־ הקרנל מנהל את התהליכונים עבור התהליך (הם אבסטרקציה של מערכת ההפעלה).

2.2.1 Kernel Level Threads

לכל ת'רד מערכת ההפעלה שומרת stack משלו ו־descriptor. ת'רדים אלו חולקים חלק מהסביבה שלהם, למשל את מרחב הכתובות והקבצים הפתוחים.

- חסרונות:
  - מחייב תמיכה של מערכת ההפעלה.
  - החלפה איטית יותר (שכן היא דורשת התערבות מצד מערכת ההפעלה).
  - כל קריאות הת'רדים (יצירה, השמדה, המתנה, החלפה ריצה) ממומשות כקריאות מערכת ומחייבות trap ־ דבר הגורם לפגיעה בביצועים.
- יתרונות:
  - מקביליות אמיתית בין תהליכונים במערכת רבת מעבדים.
  - קל לתמוך במקביליות אמיתית בין CPU ל־I/O.
  - preemptive scheduling ־ מאפשר הפקעה ברמת התהליכון.

2.2.2 User-Level Threads

ממומשים כ־library calls מכל runtime שרץ ב־user space. מערכת ההפעלה אינה מודעת לקיום הת'רדים.
כדי להחליף בין הת'רדים, עוצרים את הת'רד הנוכחי, שומרים את המצב הנוכחי של הת'רד, וקופצים לת'רד הבא. כאשר נחזור לת'רד הקודם, הוא ימשיך מהמצב שנשמר. מימשנו בתרגיל 2.
יתרונות:

- החלפת ת'רד מהירה (ללא התערבות הקרנל).
- ניתן למימוש מעל כל מערכת הפעלה.

חסרונות:

- לא מאפשרת מקביליות אמיתית (מבחינת ביצועים) עם ריבוי מעבדים.
- אין preemptive scheduling ברמת הת'רד ־ צריך $yield$ מפורש.
- קשה לאפשר מקביליות בין ה־CPU ל־I/O ־ blocking system calls מתהליך אחד חוסמות את כל התהליכונים.
- כיוון שרב התוכניות הללו מבצעות קריאות מערכת, הקרנל ממילא מעורב ויכול להחליף ת'רדים בקלות, אז למה לנסות לחסוך את ה־$trap$ (שהיינו צריכים ב־kernel level threads)?

2.3 סיכום מהיר - הבדלים בין תהליכים לת'רדים

	תהליכים	Kernel Threads	User Threads
שיתוף זיכרון	רק בקריאות מערכת מיוחדות	הכל חוץ ממשתנים לוקליים של פונקציות	
תקורה - overhead	גבוהה - החלפה יקרה בקרנל	בינונית - מהירה בקרנל	נמוכה - ביוזר מוד
הגנה אחד מהשני	✓	X	X
בעת בלוקינג של אחד אחרים יכולים לרוץ	✓	✓	X
יכולים לרוץ על מעבדים שונים בו"ז	✓	✓	X
אפשרי ללא תמיכת מ"ה	X	X	✓
מדיניות זימון שונה לכל אפליקציה - אפשרי?	X	X	✓

2.4 Pthreads

ת'רדים אלו הם Kernel-level threads.
כאמור, לת'רד יש פלאו עצמאי וניתן לתזמן אותו מכיוון והוא מתחזק:

- סטאק.
- רגיסטרים (מצב CPU).
- תכונות של תזמון (למשל מדיניות או עדיפויות).
- סט של סיגנלים *pending* וחסומים.
- מידע ספציפי לת'רד.

בסביבת UNIX:

- ת'רד משתמש במשאבים של התהליך.
- יש לו control flow עצמאי.
- משכפל רק את המשאבים החיוניים.
- יכול לשתף את המשאבים של התהליך עם ת'רדים אחרים.
- מת אם תהליך האב מת.
- הוא "lightweight" מכיוון ורב התקורה היא חלק מהיצירה של תהליך האב.

הסמן בעת קריאת קובץ הוא גלובלי לכל הת'רדים. עם זאת, כל ת'רד לעשות R/W בו זמנית, דבר שעלול לגרום לבעיות. על כן יש לשמור על קוהרנטיות המידע, וזהו אחד מהאתגרים הקשים ביותר בפני מפתחים.
thread-safeness: היכולת להוציא לפועל ת'רדים רבים סימולטנית מבלי להרוס את המידע המשותף. בהרבה מערכות הפעלה לא כל הפונקציות הן thread-safe, אך עם הזמן יש פחות ופחות כאלו.
PThread - הספריה ב-C בעזרתה יוצרים ת'רדים. מחולקת ל-3 חלקים עיקריים:

- ניהול הת'רדים.
- יצירה ועבודה עם Mutex.
- Condition variables - הכללה של מנעולים המרשה לת'רד לחכות לתנאי כלשהוא.

2.4.1 Mutex

Mutex הוא סוג של משתנה מיוחד המנוהל ע"י מערכת ההפעלה. הוא בעל שני מצבים - נעול ופתוח.
כמה ת'רדים יכולים לנסות לנעול אותו. אחד בלבד יכול לנעול, מערכת ההפעלה עוצרת את השאר, זה שנעל ממשיך לרוץ עד שמסיים, ואז הוא משחרר את הנעילה (ורק הוא כמובן יכול לשחרר, ת'רדים אחרים לא יכולים לשחרר הנעילה - מנגנון זה כאמור מנוהל ע"י מערכת ההפעלה). בשלב זה הת'רדים שוב מנסים לנעול, שוב אחד מהם מצליח לנעול ורץ ואחרים מחכים, וחוזר חלילה. יש לציין שאין סדר מסויים בו ת'רדים מצליחים לנעול. נציין שוב כי שאר הת'רדים אפילו לא יכולים לקרוא דברים שנעולים. יש למחוק את המיוטקס כאשר אין בו יותר צורך.
האחריות של מציאת קטעי הקוד שיש לנעול אותם היא של המתכנת.
אחת התקלות הטכניות שיכולות להתרחש - Deadlocks! אלו מצבים שקשה מאוד לשחזר אותם (כי הם קשורים לסדר ריצה, וכאמור אנחנו לא יודעים מהו הסדר והוא תלוי בהרבה פרטים חיצוניים). על כן - אם משתמשים ביותר ממיוטקס אחד, יש לוודא שהם לא יוצרים deadlock. ראינו למשל ב-DB שדרך אחת להמנע ממצבים כאלו היא לשמור על אותו הסדר בין המנעולים.

2.4.2 Conditional Variables

בעוד מיוטקסים ממשמשים סינכרוניזציה על ידי שליטה בגישה של ת'רד למידע, condition variable מרשה לת'רדים להסתנכרן בהתבסס על ערך אמיתי.

ללא כלי זה, המתכנת היה צריך לתת לת'רדים לבדוק שוב ושוב אם התנאי לו הם מחכים מתקיים. כלי זה מאפשר להשיג את המבוקש ללא *polling* - מונע busy wait.

ה-*lock* הוא Mutex, שמשוחרר כאשר מסיימים את השינה. לרב מטרתו - לוודא כי הת'רד שהרדים לא יסיים את הריצה.

signal מעיר רק ת'רד אחד המחכה ל-condition variable מסויים, הוא לא מחוייב כלל להעיר את הת'רדים לפי סדר כלשהוא. אם רוצים להעיר את כל הת'רדים, אפשר לעשות זאת ע"י שימוש בלופ או ע"י הפונקציה broadcast.

2.5 ריבוי תהליכים

הגדרה 2.4 מולטיטסקינג - מצב בו מספר תהליכים רצים על אותו המעבד בעזרת time sliceing.

הגדרה 2.5 מולטיפרוגרמינג - מצב בו מטפלים במספר עבודות במערכת (על אותו המעבד, או על מעבדים שונים).

הגדרה 2.6 מולטיפרוססינג - מצב בו נעשה שימוש במספר מעבדים עבור אותה העבודה או המערכת.

כאשר יש מעבד יחיד, מולטיטסקינג ומולטיפרוגרמינג הם אותו הדבר.

2.5.1 למה ריבוי תהליכים?

- תגובתיות (responsiveness) - תמיכה בהרבה משתמשים אינטרקטיביים, "important to keep users happy".
- נצילות (utilization):
  - המעבד והתקני ה-I/O יכולים לעבוד בו"ז - "יש תמורה בעד החומרה".
  - אפשר לשפר זמן תגובה אם יש job mix טוב - אולם כפי שראינו, יכולת השיפור מוגבלת.
- בו-זמניות (concurrency):
  - אינטרקציה בין תהליכים הרצים בו זמנית.
  - הרצה בו זמנית של מספר תהליכים.
  - הרצה של תהליכים ברגע בזמן שעושים משהו אחר.
- ריבוי ליבות (multi-core) - כיום כבר לא מייצרים מחשבים עם ליבה אחת. במחשב מרובה ליבות מריצים מספר תהליכים בו"ז על מעבדים שונים.

2.5.2 המחיר של ריבוי תהליכים

- תקורה - context switching overhead - מזבזים סייקלים על החלפת תהליכים.
- תחרות על משאבים משותפים עלולה לפגוע בביצועים (של תהליך נתון).
- אופטימיזציות בחומרה עובדות פחות טוב - pipeline, caching.
- סיבוכיות - שליטה ב-concurrencty, סינכרון, הקצעת משאבים, מניעת דדלוקים.

היתרונות עולים על החסרונות, ו(כמעט) כל המערכות היום הינן מרובות תהליכים.

3 תיזמונים של תהליכים ות'רדים - Scheduling

3.1 זימון במערכות Batch

מערכת batch היא מערכת עם משאב בודד.

3.1.1 מדדים

מהו "תזמון טוב"? תלוי בהרכב המשימות (תהליכים\ת'רדים) שלנו, ובמה שאנו מנסים להשיג. נגדיר מספר מדדים:

הגדרה 3.1 turnaround time\response time - זמן ביצוע כולל. מורכב משני רכיבים:

- זמן ריצה.
- waiting time - זמן המתנה (כמה זמן המשימה ממתינה לביצוע).

לעיתים מעניין אותנו זמן ממוצע. השליטה שלנו היא רק על כמה זמן ממתינה המשימה.

הגדרה 3.2 Slowdown - זמן ביצוע כולל לזמן חלקי זמן ריצה - $\frac{T_{resp}}{T_{run}}$ - פי כמה זמן לקח לתהליך לרוץ ביחס לריצה בה הוא רץ לבדו.

אנו נזניח את ה-overhead של מערכת ההפעלה, כלומר הצרכים של מערכת ההפעלה עצמה מה-CPU.
עוד דבר שמעניין אותנו כמתכנני מערכת הפעלה הוא כמה תהליכים הצלחנו להעביר. רעיון זה מועבר ע"י המושג הבא:

הגדרה 3.3 throughput תפוקה - כמה תהליכים מסתיימים ביח' זמן בממוצע = כמה תהליכים מתחילים ביח' זמן בממוצע.

כאשר יש שיוויון המערכת נקראת יציבה. זה לא קורה למשל אם כמות התהליכים שמצטברת גדולה מהתהליכים שאפשר לספק - "תור שמתפוצץ".
ההנחה שלנו במהלך כל היום הוא שהמערכת יציבה, כלומר, היא יכולה לספק את כל התהליכים שמגיעים אליה, והשאלה היא מה מבוצע קודם. אותנו יעניין כמה אלגוריתם הזימון משפיע על הגורמים השונים, ביניהם, מה ההשפעה של האלגוריתם על התפוקה.
מתי אפשר לשפר את התפוקה? אם התור חסום, אלגוריתם הזימון יכול לשפר את התפוקה רק ע"י הקטנת התור.

הגדרה 3.4 utilization נצילות - אחוז הזמן בו ה-CPU עסוק. הוא פונקציה של התקורה והתפוקה.

כאן אלגוריתם הזימון צריך <u>לא לעבוד</u> כאשר התור לא חסום, מכיוון וכשהוא עובד הוא מנצל את ה-CPU.

הגדרה 3.5 fairness - הוגנות.

הבעיה עם המושג - מה שהוגן לאחד לא הוגן לאחר. המטרה היא להקטין את הסתירות בין ההוגנות כלפי התהליכים השונים כמה שיותר.

הגדרה 3.6 הוגנות חלשה - כל מי שמבקש שירות בסופו של דבר מקבל, כלומר אין הרעבה starvation.

הגדרה 3.7 הוגנות קצת יותר חזקה - לא מחכים זמן לא חסום.

הגדרה 3.8 הוגנות חזקה - כולם מקבלים הזדמנויות שוות לרוץ.

היא לא מכסה את כל הדברים שאפשר, אבל זוהי הפייריות שנתייחס אליה כנייר לקמוס לבדיקת היעילות שלנו. לא ניתן להגיע באמת למדד הזה, אך ננסה להתקרב ככל האפשר אליו.
ישנו מדד נוסף שלעיתים מאוד חשוב, למשל כאשר מתכננים מערכת שצריכה להעביר שיחות טלפון דרך הרשת. לא צריך את כל המשאב, אבל צריך לוודא שהאודיו מקבל מספיק משאבים כל הזמן. בנושאים כאלו, שה-realtime חשוב, מתייחסים למושג הבא:

הגדרה 3.9 predictiability - יכולת חיזוי - עד כמה יודעים להעריך כמה זמן משימה תחכה בתור, או משימות מסוג מסויים.

נתמקד במדד - זמן המתנה ממוצע (כשאנחנו לא יודעים משהו יותר טוב, או כשאנחנו רוצים באופן כללי להעריך את המערכת). כאמור, לא בכל המקרים הוא המדד הכי טוב, אבל במידה ואין לנו מידע נוסף על דרישות המערכת זהו מדד שטוב להתייחס אליו.
אנו נתמקד במדד - זמן המתנה ממוצע.
ישנם שני סוגי זימונים:

הגדרה 3.10 זימון $offline$ - תכנון מראש בידיעה מלאה. לא מציאותי, אבל נותן אינטואיציה ובסיס להשוואה - baseline.

הגדרה 3.11 זימון $online$ - תכנון במהלך הריצה ללא ידיעה מראש.

מערכת רגילה היא שילוב של השניים.

אלגוריתם 1 אלגוריתם זימון אונליין - SRT - Shortest Remaining Time

מניחים כי זמן הריצה של תהליך ידוע עם הגעתו. אזי, כאשר מגיע תהליך חדש:

- אם זמן הריצה של התהליך החדש קצר מיתרת זמן הריצה של התהליך שרץ כרגע - יבצע preemtion יפסיק את התהליך שרץ, ויריץ את התהליך החדש.
- אחרת, יכניס אותו לתור הממוין לפי יתרת זמן ריצה.
- כשמסתיים תהליך, יריץ את התהליך מראש התור.

3.1.2 זימון אופליין

נניח שאנו מקבלים את כל המשימות על ההתחלה. אזי המשימה שלנו היא למעשה לסדר את התהליכים הנתונים בסדר נכון.
ה־scheduler עובד כתוכנית סדרתית. אין סיבה להפסיק תהליך בזמן שהוא רץ - preeemption, שכן לא נרוויח דבר בזמן ריצה ממוצע אם נפסיק ונמשיך תהליך בהמשך, רק נאסוף עוד overhead.
אלגוריתמים לזימוני אופליין:

- FCFS - First Come First Served: האלגוריתם הפשוט ביותר - סדר מתן ה־CPU זהה לסדר בקשתו . המימוש: ע"י FIFO - הרצת תהליכים אחד לאחר השני, לפי סדר כניסתם לתור.
  - הבעיה: אם מקבלים את המשימות כך שמשימה הארוכה ביותר מתקבלת ראשונה, נקבל זמן ממוצע ארוך: אפקט השיירה - הרבה תהליכים קטנים מחכים מאחורי תהליך גדול יחיד.
- פתרון לבעיה - Shortest Job First (SJF) - נעדיף את המשימות הקצרות לפני הארוכות: מבטיח שבמקום הקטנים יחכו הרבה, נתחיל עם המשימות הקטנות, ואז המשימות הארוכות בממוצע יחכו פחות. עם זאת, אותה כמות משימות שנכנסה היא הכמות שיוצאת, כלומר, אלגוריתם זה לא השפיע על התפוקה. נשים לב כי זהו אלגוריתם offline בלבד, שכן יש לדעת מראש את כל המשימות ואת אורכן.

3.1.3 זימון אונליין

עדיין יש משאב יחיד, אולם המשימות לא נתונות אלא מגיעות אונליין - תוך כדי ריצה, וכן לא ידוע לנו מתי הן מגיעות (לו היה ידוע לנו, המקרה היה יותר קל ודומה לאופליין).
הערה - אם לא ניתן להפסיק תהליכים, מדובר בבעיה מאוד סבוכה (NP − complete למספר מעבדים). אנו עם זאת נעסוק רק במקרה בו מותר $preemption$ - כלומר, היכולת להפסיק תהליך במהלך ריצתו ולהחליף את התהליך הרץ בתהליך אחר.
אלגוריתם לזימון אונליין צריך:

- להתאים את הזימון לתנאים משתנים - שכן מגיעים תהליכים חדשים.
- לפצות על חוסר ידע - למשל זמן ריצה של תהליך.

preemtion מאפשר למערכת ההפעלה לשפר הוגנות (אם תהליך ארוך מאוד רץ ומגיע תהליך קצר יותר תוך כדי).

ניתן להוכיח שה- SRT מבטיח זמן המתנה ממוצע אופטימלי, בתנאי שמזניחים את התקורה של ה־context switch בשל ה־preemption.
בעיה עם האלגוריתם - הרעבה: יגרום להרעבה במערכת שעובדת בעומס גבוה (כאשר מגיעות הרבה משימות קצרות בזמן קצר), משימות ארוכות לעולם לא יטופלו. אם המערכת היא כזו שהניצולת של ה־CPU היא לא 100%, אזי התהליכים שהגיעו קודם הסתיימו, כלומר לא תהיה הרעבה.
אבל, אלגוריתם זה גם אינו ישים במערכת אינטרקטיבית רגילה, וכמו כן, לא responsive - לא נותן להרבה משתמשים הרגשה שהתהליכים שלהם רצים בו־זמנית.

3.2 זימון במערכות אינטרקטיביות

המאפיינים של המערכת:

- לא ניתן לחזות מראש את זמן הריצה של התהליך.
  - preemption תקופתי מונע מתהליכים ארוכים לתפוס את המעבד לזמן ממושך.
- המערכת צריכה להגיב למשתמשים גם במהלך הריצה של תהליך.

אלגוריתם 2 אלגוריתם Round Robin

- לכל תהליך timeslice קבוע, אם לא הסתיים עד סוף הזמן המוקצב, מוציאים את התהליך הנוכחי, מכניסים אותו לסוף התור, ועוברים לתהליך הבא בתור.
- גם תהליך שמסיים או עובר למצב $wait$ לפני סיום פרוסת הזמן מפנה את ה־CPU.
- כשמגיע תהליך חדש, מכניסים אותו לסוף התור, כדי לנסות לשמור על הרצף שהיה קודם לכן (אם היה נכנס לתחילת התור, היינו עלולים לגרום להרעבה).

- לא רק זמן סיום התהליך קובע - גם תגובתיות חשובה!
- preemtion תקופתי מאפשר להגיב להרבה תהליכים בו זמנית.

- תהליך במערכת כזו משתמש ב־CPU וב־I/O לסירוגין.

הגדרה 3.12 קטע CPU בין שתי פעולות I/O נקרא CPU burst.

מהסיבות מעלה, תזמון במערכת אינטרקטיבית תמיד יבצע preemtion. כמו כן, נניח שזמני ההגעה אינם ידועים, וזמן החישוב אינו ידוע.
מכיוון וזמן ביצוע כולל במערכת זו כולל גם את הזמן של פעולות ה־I/O - לא בשליטת ה־scheduler של ה־CPU.

3.2.1 מדדים במערכת אינטרקטיבית

על כן, כאן המדדים הינם:

1. זמן תגובה.
2. responsivness.
3. תפוקה.

הגדרה 3.13 זמן תגובה response time - זמן ביצוע כולל של CPU burst (לא תהליך שלם).

אבל גם "זמן תגובה ממוצע לא תמיד מספיק טוב, כי הממוצע לא אומר הרבה". על כן המדד השני הוא:

הגדרה 3.14 responsiveness תגובתיות - זמן תגובה טוב (לפחות סביר) לכולם - מדד זה מביע את התחושה של המשתמש.

הגדרה 3.15 תהליכים compute bound הם תהליכים שמנצלים הרבה CPU בין פעולות I/O.

הגדרה 3.16 תהליכים I/O bound הם תהליכים המנצלים מעט CPU בין כל שני קטעי I/O (למשל מעבד תמלילים).

אזי:

- כדי לשפר את זמן התגובה צריך להעדיף תהליכים עם CPU bursts קצרים.
- כדי לשפר תפוקה צריך להעדיף גם תהליכים עם CPU bursts קצרים (כדי לנצל את ה־I/O באופן יעיל).
- איך זה משפיע על זמן ביצוע כולל של תהליכים? כביכול צריך לתת לתהליכים עם מעט I/O עדיפות נמוכה. עם זאת, תהליכים כאלה בדר"כ נוטים להיות קצרים יותר. ולכן זמן ביצוע כולל ממוצע יפגע.

כיוון וה־CPU אינו ניתן לחלוקה, נחלק את הזמן ליחידות בגודל קבוע - quantum/time slice, כדי להתקרב כמה שיותר לחלוקה של ה־CPU. הבעיה היא שכשעושים את זה ה־overhead עולה, לכן מעשית לא נותנים זמנים קטנים מאוד.

3.2.2 אלגוריתם Round Robin

דומה ל־FCFS, אך משתמש ב־preemption. תור התהליכים המוכנים - ready, הוא מעגלי, וכך עוקפים את הבעיה שאיננו יודעים מהו האורך של כל תהליך.

בחירת גודל ה־quantum - משפיע על התקורה, ולכן יש לבחור גודל אופטימלי (ב־unix ההקצבה היא 100msc) - קצר בשביל לשמור על הוגנות, אך גדול מספיק כך שה־context switch קצר ביחס אליו.

תכונות ה-RR:

- זמן תגובה מהיר.
- זמן המתנה ממוצע גבוה מ-SRT.
- מניח שכל התהליכים חשובים באותה המידה.

3.2.3 Priority Scheduling

העקרון: לא כל התהליכים חשובים באותה המידה. מדיניות זו קובעת עדיפויות שונות לתהליכים שונים.

- עדיפויות סטטיות: קבועות לאורך כל ריצת התהליך.
  - עדיפות לא נכונה של קרנל יכולה לגרום לדדלוק (לא נדון בזה).
  - ב-Unix - פקודת nice להורדת עדיפות, וכן יש עדיפות ל-kernel.
- עדיפויות דינמיות: עפ"י צריכת משאבים.
  - עדיפות למי שצורך מעט CPU, כלומר I/O bound.
  - למשל מימוש עיקרון ה-SRT עבור CPU bursts.
  - בשביל להקציב עדיפויות בצורה נכונה, יש צורך בחיזוי העתיד מתוך העבר.

שיטת חיזוי - שערוך גודל ה-cpu burst של תהליך נתון ע"י מונה דועך אקספוננציאלית:

$$E_{n+1} = \alpha T_n + (1-\alpha)\, E_n,\ 0 \le \alpha \le 1$$

כאשר T_n הוא האורך של ה-burst ה-n-אי, ו-E_{n+1} הוא האורך שנצפה שיקח ל-burst הבא.
שיטה פשוטה יותר בה משתמשים ב-Unix - מונה שגדל בכל תהליך רץ בכל פסיקת שעון, חלוקה ב-2 של כל המונים באופן תקופתי - דעיכה אקספוננציאלית לפי הזמן שחלף, לא לפי מספר ה-bursts. בוחרים להעדיף תהליכים עם ערכי מונה נמוכים יותר.

3.2.4 תור עדיפויות - Priority Queue

זהו מנגנון זימון המיישם מדיניות של זימון מבוסס עדיפויות. העדיפויות יכולות להיות סטטיות או דינמיות. הזימון יכול להיות preemptive או לא - preemptive יזמן תהליך מחכה בעל עדיפות גבוהה יותר מהתהליך שרץ.
נרצה לתת עדיפות לתהליכים עם עדיפות גבוהה על פני עדיפות נמוכה, אך לדאוג שלא תהיה הרעבה. הפתרון לכך - aging - עם הזמן לתהליך שמחכה מעלים את העדיפות.
אנו צריכים מנגנון שיעזור לנו לנהל עדיפויות שונות, ובכל עדיפות לנהל את התהליכים שמחכים - על כן נחלק את התהליכים לתורים - multi level queues, ולממש מדיניות פרטית בכל תור בנפרד (באחד RR, באחר מנגנון דינמי שונה, וכו').
priority scheduling - מריצים את כל התהליכים בתור בעדיפות גבוהה במלואם, ואז ממשיכים לתור בעל עדיפות נמוכה. מדיניות שיוויונית יותר - הקצאה שווה של CPU לכל תור. עוד קריטריון לקביעת עדיפות - סוג התהליך שרץ.
צורת המימוש - Multi level feedback queues:

- חלוקה לתורים, לכל תור יש עדיפות.
- תהליכים יכולים לעבור בין תורים באופן דינאמי.
- הפרדה לרמות בהתאם לשימוש ב-CPU.
  - תהליכים שהם I/O bound מקבלים עדיפות גבוהה.
  - ניתן לעקוב אחר הקשות המקלדת כדי להעלות עדיפות של תהליכים אינטראקטיביים.
  - אם תהליך הוא CPU-bound, אז ככל שזמן הביצוע שלו מתארך, עדיפותו יורדת.
- בתוך כל רמה אפשר להשתמש באלגוריתם עדיפות שונה.
- הקצאת CPU לכל תור - בדר"כ quantum קטן בעדיפויות הגבוהות.

ב-unix קלאסי:

- multi-level feedback.

- תור לכל רמת עדיפות.
- RR בכל תור, אותו quantum בכל התורים.

- ב־process table יש שדה עדיפות - פונקציה של המחלקה אליה שייך התהליך, וזמן הריצה עד כה:

$$pri = cpuuse + base + nice$$

- base מפריד בין עדיפויות סטטיות בין ה־user ל־kernel, ובין סוגי פעולות\שירותים בקרנל.
- cpu_use - נקבע באופן דינמי.

- הקרנל מעדכן עדיפויות באופן הבא:

- לתהליך שמחכה - העדיפות עולה.
- העדיפות של תהליך יורדת במהלך ריצתו בהתאם לאורך הריצה.
- שעון מעדכן עדיפויות של כל התהליכים ב־user-mode אחת ל־10msec.

מה נותנת לנו השיטה הזו?

- יש עדיפות לתהליכים אינטרקטיביים (משלבים עיבוד קצר עם המתנה לתגובה של המשתמש).
- אם כל התהליכים בעלי אותו פרופיל, הקצאת זמן הריצה ביניהם תהיה שיוויונית.

3.3 לסיכום

- אלגוריתמי אופליין קלים לניתוח, נותנים אינטואיציה.
- רעיון חשוב - הרצת תהליכים קצרים לפני ארוכים:

- שיפור זמן המתנה ממוצע.
- שיפור ניצול התקני I/O (במקרה של CPU bursts קצרים).
- פגיעה בהוגנות... עד כדי הרעבה. פתרון להרעבה: aging.
- דורש חיזוי העתיד: העבר הוא נביא טוב ב־workload טיפוסי.

- אנחנו מנסים להגיע ליעדים סותרים אחד את השני. פתרון - שימוש בכלים שונים שמונעים הרעבה.
- אין אפשרות אמיתית לדעת מה כל סוג של תהליך יצרוך, לכן משתמשים בכלי שיעזור לנו להעריך את העתיד על בסיס העבר.
- בעולם האמיתי מחפשים את שביל הזהב בין ביצועים, הוגנות, ופשטות היישום, בהתאם לייעוד המערכת - טלפונים סלולריים, מחשבים אישיים, מחשבים ייעודיים, ועוד.

4 מקביליות - Concurrency

מקביליות טובה עבור משתמשים - מאפשרת עבודה על אותה הבעיה, ביצוע של תוכנות סימולטנית, ביצוע משימות ברקע...
עם זאת, היא מחייבת את מערכת ההפעלה לתמוך באינטרקציות בעייתיות:

- גישה למסדי נתונים משותפים.
- סכנה לדדלוק עקב תחרות על המשאבים.

4.1 הגדרת התנהגות תוכנית מקבילית

הגדרה 4.1 מירוץ $(race)$ הוא מצב בו זימון הפעולות (מי רץ בדיוק מתי) משפיע על התוצאה הסופית.

ההגדרה צריכה להיות אבסטרקטית - ללא התחשבות במהירויות המעבדים, ;זימון תהליכים, וניהול גישה למשתנים משותפים.

הגדרה 4.2 ריצה היא סדרה של מצבים בהם התוכנית עוברת.

לתוכנית מקבילית יש ריצות שונות, כל אחת בהתאם לזימון. התנהגות של תכנית מקבילית מוגדרת בעזרת התכונות שלה.

הגדרה 4.3 תכונה $(property)$ היא נוסחא בוליאנית (פרדיקט) על ריצה. נגיד כי תוכנית מקיימת תכונה, אם כל ריצה שלה מקיימת את התכונה.

ישנן שני סוגי תכונות:

הגדרה 4.4 נגיד שתכונה היא תכונת $safety$ אם היא מגדירה משהו נכון תמיד, כלומר, כל מצבי המערכת "בטוחים".

הגדרה 4.5 נגיד שתכונה היא תכונת liveness, אם היא מגדירה משהו שבסופו של דבר מתרחש.

4.2 בעיית הקטע הקריטי

אנו רוצים לאפשר לתהליכים לבצע פעולות המורכבות מיותר מפקודה אחת באופן אטומי. כלומר, שיראה כאילו אף תהליך אחר לא מבצע פקודות במהלך הפעולה. בפרט, נרצה לוודא שאף תהליך לא יראה "מצב ביניים", ולא משנה משתנים שהפעולה קוראת תוך כדי ריצתה.

הבעיה - לא כל הקוד של תהליך מתבצע באופן אטומי (אחרת לא היינו מבצעים מקביליות כלל).
על כן, מבדילים בין קטעים קריטיים (critical section) שצריכים להתבצע באופן אטומי, לבין שאר הקוד - remainder.
תפקידנו כמתכנתים - לבחור את הקטעים בקוד שהם קטעים קריטיים. בדר"כ מדובר בגישה למשאב משותף - כמו כן, בדר"כ לא נעשית גישה כזו מחוץ לקטע קריטי.
מה מיוחד ב"קטע קריטי"?

- מי שרוצה להכנס אליו, צריך לבקש רשות לבצע כניסה - entry.
- אחרי שתהליך סיים לבצע את הקטע הקריטי, הוא מבצע יציאה - exit.

פתרון הבעיה - מימוש של ה־entry וה־exit. מימוש זה בדר"כ מסופק ע"י מערכת ההפעלה. אסור להניח כלום על קצבי חישוב יחסיים או הזימון.

4.2.1 הגדרת הבעיה - תכונות

על מנת לפתור את הבעיה, נרצה כי הפתרון שלנו יקיים את התכונות הבאות:

הגדרה 4.6 Mutual exclusion (מניעה הדדית) - לכל היותר תהליך אחד נמצא בקטע הקריטי בזמן נתון.

הגדרה 4.7 progress (התקדמות) - אם אף תהליך לא נמצא בקטע הקריטי ויש תהליכים הרוצים להיכנס, אזי בסופו של דבר יהיה תהליך שיכנס.

תכונה אופציונלית שהפתרון יכול לקיים:

הגדרה 4.8 bounded waiting/fairness (הוגנות) - אם תהליך p מבקש להכנס לקטע הקריטי, אזי יש חסם על מספר הפעמים שתהליכים אחרים נכנסים מאז ש־p מבקש ועד שהוא נכנס.

קיום תכונה זו מונע הרעבה.

הגדרה 4.9 generality - (כלליות) - הפתרון עובד עבור N תהליכים.

כאשר אין התקדמות, הדבר נובע מכך שהריצה הגיעה לאחד משני המצבים הבאים:

הגדרה 4.10 deadlock הוא מצב בו אף תהליך לא יכול לבצע צעד שיביא להתקדמות.

הגדרה 4.11 livelock הוא מצב בו תהליכים כל הזמן ממשיכים לבצע צעדים, אך הם לא מובילים להתקדמות.

4.3 פתרונות לקטע קריטי - ללא תמיכת חומרה

4.3.1 האלגוריתם של פיטרסון

פתרון זה איננו תלוי מערכת הפעלה, שכן אין שום שימוש בה. בתרגול מובאת ההרחבה של אלגוריתם זה ל־N תהליכים.

הרעיון המרכזי של האלגוריתם ל־2 תהליכים: שימוש במערך משותף של flags לציון "רצון" של תהליך להכנס לקטע הקריטי, ומימוש משתנה אינט משותף לציון "תור מי להכנס לקטע" - אם התהליך השני רוצה להכנס, ותורו, המתן. אחרת, הוא יכנס לקטע הקריטי, ובסופו ישנה את ה־flag שלו ל־$false$.

הרעיון המרכזי של האלגוריתם ל־N תהליכים: ישנם שני מבני נתונים משותפים:
$q[n]$ - כאשר $q[i]$ הוא השלב שבו פרוסס i נמצא בסולם.
$turn[n-1]$ - $turn[j]$ אומר איזה פרוסס בשלב j ישאר במקום ולא יתקדם - "איזה פרוסס יחזיק את המקל".
יוצאים מהלולאה הפנימית בשני מקרים:

1. כל הפרוססים האחרים התקדמו וסיימו, כך שאין אף אחד אחר בסולם שהוא בשלב מתקדם מאיתנו.
2. אם פרוססים אחרים רצים ו"מעבירים את המקל" למישהו אחר.

כשיוצאים מהלולאה הפנימית מתקדמים בלולאה החיצונית, וכל פעם שזה קורה, בעצם מתקדמים בסולם. כאשר מגיעים לשלב ה־n מגיעים ל־critical section.
אלגוריתם זה מקיים את כל הקריטריונים: Mutual exclusion, progress, fairness and generalty.
מה שיפה בשיטה הזו - היא אינה משתמשת במערכת ההפעלה. עם זאת, היא עושה busy wait, שזה מאוד לא יעיל.

4.3.2 Model Checking

הוכחה פורמלית של קיום התכונות הדרושות הינה ארוכה ומסובכת, אפילו לאלגוריתמים פשוטים יחסית (כגון האלגוריתם של פיטרסון).
Model Checking היא דרך נפוצה לאימות אוטומטי של מערכות חמרה ותוכנה - בודקת אם תוכנית מסוימת (חומרה או תוכנה) מקיימת ספסיפיקציה הנתונה כנוסחא במספר רב של מצבים אליהם התוכנה יכולה להגיע.

4.3.3 פתרון ל־n תהליכים - אלגוריתם המאפיה

הרעיון: כל תהליך נכנס ו"לוקח מספר" - $ticket[i]$ (הערך שהוא לוקח הוא הערך המקסימלי שקיים כרגע+1), התהליכים נכנסים לקטע הקריטי לפי סדר עולה של מספרים. אולם, הקצאה זו לא נעשית באופן אטומי, ולכן כמה תהליכים יכולים לקחת את אותו המספר (נסיון ראשון). על כן, נבדוק זוגות של $(ticket[i], i)$, ונחליט על הסדר הלקסיקוגרפי הבא:
$(ticket[j], j) < (ticket[i], i)$ אם $ticket[j] < ticket[i]$ (כלומר, לקח מספר קודם בתור), או אם שני המספרים זהים, וגם מתקיים $j < i$.
הבעיה במימוש השני - תהליך לא יודע אם תהליך אחר רוצה גם לקחת מספר, ומכיוון ופעולת לקיחת מספר איננה אטומית, יכולה להיות הפרה של ה־mutual exclusion.
כיצד מתקנים בעיה זו?
לא נהפוך את הפעולה לאטומית, אלא נוסיף סימן - ממשים מערך נוסף של אינטים בשם choosing. מוסיפים תנאי בלולאה - תהליך בקטע ההמתנה צריך גם לבדוק אם מישהו עכשיו לוקח מספר. אם כן, הוא מחכה עד שהתהליך הזה יסתיים, ורק אז בודק האם יש מישהו לפניו בתור.
פתרון זה מבטיח לנו Mutual exclusion, progress, וגם את התכונה השלישית - bounded waiting.

הגדרה 4.12 משתנה מסוג $safe$ הוא משתנה שאם אין בו בו־זמניות, קריאה מחזירה את הערך האחרון שנכתב, ואם יש בו כתיבה בו־זמנית לקריאה, הוא יכול להחזיר ערך שרירותי.

4.4 פתרונות נתמכי חומרה

4.4.1 Test-and-set-lock

הגדרה 4.13 Test-and-set-lock היא פעולה המשנה משתנה ל־true ומחזיר את ערכו הקודם באופן אטומי.

הפתרון עצמו לא מאפשר במנגנון הבסיסי לומר כלום על מספר הממתינים וכמה זמן הם ממתינים, לכן הרעבה אפשרית. הוא כן עובד למספר כלשהוא של תהליכים. כמו כן, פתרון זה לא מתגבר על הבעיה של busy wait.

4.4.2 הרחבה של TSL לזמן המתנה חסום

הוספה של מערך waiting (כאן ההנחה היא שלכל התהליכים יש את אותה העדיפות) שמציין אם התהליך i מחכה.
בלולאה הראשונה - כותבים את הערך של TSL למשתנה מקומי, ומחכים. לכאורה אנחנו מחכים למשתנה מקומי ול-TSL.
בהתחלה אחד יכנס (גם אם הגיעו כמה לאותו מקום יחדיו), שכן אחד יקבל ערך $false$ ל-TSL.
ביציאה דואגים ל-bounded wait - עוברים על ה-waiting, ומחפשים מישהו שמחכה או את עצמנו:

- משנים את ה-$lock$ אם זה אנחנו (מהקוד עברנו על כל השאר ואף אחד מהם לא מחכה).
- אם מצאנו מישהו אחר, משנים לו את הערך שלו במערך waiting כך שיוכל להכנס (וכך לא תווצר הרעבה) - כל מי שממתין נכנס תוך מקסימום n תורות. כמו כן מכיוון ולא שחררנו את המנעול אף תהליך אחר לא יוכל להכנס לקטע הקריטי.

4.4.3 פתרונות בעזרת פעולות אטומיות בחמרה - סיכום

יתרונות:

- פשוטים.
- תומכים במספר כלשהוא של תהליכים.

חסרונות:

- busy waiting + priority inversion.
- או שתיתכן הרעבה, או שהפתרונות לא כ"כ פשוטים.
- לא portable בין חומרות שונות, ולכן לא מתאימים ברמת משתמש.

כמו כן, לאפליקציות יש צרכי סנכרון נוספים מעבר לקטעים קריטיים, למשל סדר בין פעולות.
נרצה כלי שיהווה אבסטרקציה ברמה גבוהה יותר:

- קלה לשימוש.
- תספק ממשק אחיד מעל חומרה כלשהיא.
- תתמוך בצרכי סנכרון שונים.
- אם מערכת ההפעלה תומכת במימוש, אז נשתמש בה.
- המימוש יוכל למנוע busy wait ו-priority inversion.

כך מגיעים לפתרון הבא:

4.4.4 Semaphore - פתרון נתמך חומרה

הגדרה 4.14 סמפור הוא abstract data type המאותחל למספר שלם אי שלילי, המשותף למספר תהליכים, ומוגדרות עליו שתי פעולות שנתמכות ע"י החומרה כך שהן נעשות בצורה אטומית:

- הפחתה - $P(v)$ ("פחות") - אם הערך קטן שווה מאפס חוסם, אחרת מוריד אחד - כלומר "תופס מנעול".
- הוספה - $V(v)$ ("ועוד") - מוסיף אחד לערך, כלומר "משחרר מנעול, ואם יש תהליכים חסומים, משחרר אחד.

סמפור מבטיח מימוש קטע קריטי בצורה נכונה, מכיוון והפעולות הללו כאמור נעשות בצורה אטומית. כמו כן, בשל כך יש progress, שכן כאשר תהליך משחרר את הסמפור, הוא נותן לתהליך אחריו להכנס לקטע.
סמפור כללי נקרא גם "סמפור סופר". סמפור בינארי הוא בעצם mutex (במקרה זה הסמפור מקבל רק שני ערכים - 0 ו-1). נקרא גם "מנעול".

מימוש סמפור סמפור מורכב מ:

- משתנה אינט.
- רשימה\תור list של תהליכים ממתינים.

סדר הטיפול הוא בשליטת מערכת ההפעלה - יכול להיות הוגן, יכול להתחשב בעדיפויות, וכו'.
בספריות המממשות מנעולים כיום, נוטים לשלב בין שני המנגנונים להמתנה: תחילה מנסים לתפוס מנעול ברמת המשתמש (busy wait), ואם אחרי מספר פעמים לא הצליח להתעורר, ממשים ברמת החומרה - התהליך נכנס לרשימת המתנה והולך לישון.

אילו בעיות סמפור פתר?

- אבסטרקציה נוחה למתכנת.
- אין הרבה בזבוז ב־*cpu*.
- יכולה להיות הרעבה (אבל גם בזה אפשר לטפל).
- האבסטרקציה היא ברמת התוכנה, לכן אין שום בעיה שהדבר יעבוד בכל מערכת הפעלה (אם היא תומכת זה רק יקל על הפעולה ב־*cpu*, אם לא עדיין יתקיים רק יעלה קצת יותר).
- מונע (או לפחות מפחית מאוד) busy wait.

לעיתים סמפור לא מספיק - יש פתרון עם אבסטרקציה גבוהה יותר.
בעיה בסמפור - התוכניתן צריך להזהר, השימוש במנגנון לכשעצמו לא מבטיח חוסר טעות. נרצה מנגנון שיספק לנו את ההגנה הזו.

4.4.5 מוניטורים

כלי כזה הוא מוניטור - אבסטרקציה ברמה גבוהה יותר. השתמשנו בכלי זה באחד התרגילים.

הגדרה 4.15 מוניטור הוא לכלי להגדרת abstract data type משותף לכמה ת'רדים - זהו אובייקט עם משתנים לוקליים ופונקציות גישה (methods). הגישה אליו רק דרך הפונקציות. כל גישה לאובייקט תגרום לחסימה.

כלומר, ביצוע של כל method הוא אטומי, או "קטע קריטי". על כן התוכניתן לא צריך לזכור לנעול בכל גישה - הנעילה מתבצעת במימוש המוניטור.
האובייקט יכול לכלול משתני סנכרון מיוחדים מטיפוס condition, עליהם מוגדרות פעולות מיוחדות:

- wait() - להמתין (תמיד ממתין).
- signal() - להעיר תהליך ממתין אחד (אם יש כזה).

לאחר ביצוע signal תהליך חייב לצאת מהמוניטור.
נשים לב כי כאן סדר הפעולות חשוב:

- אם wait קורה לפני signal, signal מעיר את הממתין.
- אם signal קורה לפני wait הוא לא מעיר אף אחד, וה־wait ממתין.

הרחבה של הסיגנל - פעולת broadcast מעירה את כל מי שממתין - הממתינים כמובן ירוצו סדרתית, לא בו זמנית.

הבעיות של מוניטורים

- בדר"כ לא נתמך ברמת מערכת ההפעלה.
- יש תמיכה במשתני condition בספריית הת'רדים.
- לא יעיל במקביליות, בגלל החסימה.

נרצה מנגנונים יעילים יותר (למשל אי אפשר לממש שם את הקוראים\כותבים - אין מקביליות אפילו בין שני קוראים).

4.5 בעיות קלאסיות של סנכרון

4.5.1 Bounded-Buffer Problem

יש באפר ציקלי שיכול להחזיק עד N איברים. משמשים בו בצורה ציקלית, כלומר, כאשר מסיימים לכתוב על הבאפר, חוזרים לכתוב מההתחלה, כנ"ל לגבי הקריאה. הבאפר משותף. המטרה: לוודא שהכותב לא דורס מידע שהקורא עדיין לא קרא, ושהקורא יקרא מידע מעודכן.
לפתרון בעיה זו, משתמשים בשלושה סמפורים:

- semaphore mutex - מאותחל לערך 1, מגן על האינדקס לבאפר.
- semaphore full - מאותחל לערך אפס, מציין כמה תאים מלאים.

- semaphore empty - מאותחל לערך N, מציין כמה תאים ריקים.

איך תתבצע כתיבה?
פניה ל-empty לבדיקה אם יש תאים ריקים. אם יש, פונה למנעול לנעילה, רושם, משחרר את המנעול, ומוסיף ל-full.
איך תתבצע קריאה?
בצורה דומה - מוחק מה-full כשיהיה תא שניתן לקרוא אותו, נועל, קורא ומוחק אותו, משחרר את המנעול, ומוסיף ל-empty.
אם יש יותר מ-producer ו-consumer אחד, זה עדיין יעבוד!

4.5.2 בעיית הקוראים-הכותבים

נתון מבנה נתונים שמספר תהליכים רוצים לכתוב ולקרוא אליו סימולטנית. מותר שיותר מאחד יקרא (כי זה לא יסתור שום דבר). כותבים לא יכולים לעבוד בו זמנית.
פתרון ראשון: נרצה לספור כמה קוראים יש בכל רגע נתון, אם אין אף אחד שקורא אפשר לכתוב. המנעולים בהמשך נשתמש הם:

- semaphore mutex שיגן על מספר הקוראים.
- semaphore write שמוודא שרק אחד כותב.

כתיבה פשוטה. קריאה: חלק ראשון מוגן ע"י מנעול כדי לוודא שלא יתכן ששני קוראים חושבים בו זמנית שהם ראשונים, ואז מנסים לקחת את המשאב של הכותב. הקורא האחרון משחרר את המשאב של הכותב.
הבעיה: הכותבים עלולים לרעוב, אם יש כל הזמן קוראים (כי הם כל הזמן יפנו למשאב).
פתרון שני: אם כותב מחכה, לא יוספו עוד קוראים, הוא יחכה עד שהם יסיימו, ואז יכתוב. כיצד?

- semaphore read שמאותחל לאחד - מונע מקוראים גישה כאשר כותב מעוניין לכתוב.
- writecount שמאותחל לאפס - סופר כמה כותבים מחכים.
- semaphore write_mutex שמאותחל לאחד - מגן על העדכון של הקודם.
- semaphore mutex מחליף שם ל-read_mutex.
- queue semaphore - משתמשים בו רק בקטע הקוד הקורא.

מדוע $\mathrm{P}(queue)$ בתחילת הקורא?
אם יש הרבה קוראים שמחכים, ואז כותב מגיע, בלי זה הוא יצטרך לחכות שכל הקוראים יתעוררו (כולם יגשו ל-$\mathrm{P}(read)$ ויגדילו אותו, והקורא יצטרך לחכות שכולם יסיימו לפעול ויורידו אותו לאפס כדי שיוכל לכתוב). אנו רוצים לוודא שאף פעם לא יהיה יותר מקורא אחד יחכה על ה-$\mathrm{P}(read)$, ואז הכותב יצטרך לחכות רק לקורא אחד, ולא לכל הקוראים.
פתרון זה נותן עדיפות לכותבים, ואף עלול להרעיב את הקוראים. אולם, הרעבה זו לא מפריעה לנו, כי הרעבת הקוראים משמעה שהכותבים עובדים אחד אחר השני, ואין קורא בין לבין. במקרה זה, פשוט אין מספיק זמן cpu לכולם, כלומר המערכת בעומס יתר.

4.5.3 Dining-Philosophers Problem

כפי שהכרנו ב-DB - 5 פילוסופים ישובים סביב שולחן עגול, מול כל אחד קערת אורז, צ'ופסטיק אחד בין כל שני פילוסופים, semaphore לכל צ'ופסטיק. נרצה לדאוג שכל הפילוסופים יוכלו לאכול (בשביל לאכול פילוסוף צריך להחזיק שני צ'ופסטיקס בו זמנית), ולא יהיה דדלוק. פילוסוף לא יכול להרים שני צ'ופסטיקים בבת אחת.
לבעיה זו אי אפשר למצוא פתרון סימטרי - כל אלגוריתם סימטרי יוביל לכך שאו שכולם יאכלו, או שאף אחד לא יאכל.
כיצד אפשר לפתור? מספר דרכים:

- בעזרת busy wait (לתת לאחד לאכול, האחרים מחכים, וכו') - אבל נרצה להמנע מ-busy wait.
- עוד פתרון - חלק יתחילו משמאל ואז מימין, חלק יתחילו מימין ואז שמאל, ואז לא יהיו דדלוקים.
- use a waiter - ניהול מלמעלה, דואגים שלפילוסוף אחד יתנו שני צ'ופסטיקים בו זמנים.
- להרשות לכל היותר 4 פילוסופים סביב השולחן.
- לתת להם עוד צ'ופסטיק.
- אלגוריתם רנדומי (להמן-רבין): הפילוסופים מרימים את הצ'ופסטיק, אם לא מקבל שניים, כל אחד מחכה זמן המוגרל רנדומית ספציפית ביחס אליו, ואז משחרר את שניהם. הסתברותית אם נחכה מספיק זמן לא יהיה דדלוק.

4.6 טרנזאקציות

הבעיות עד כה:

- סמפורים יכולים ליצור דדלוקים אם השימוש של התוכניתנים בהם אינו זהיר. היינו רוצים שדבר כזה לא יקרה. כמו כן, היינו רוצים מנגנון יותר אבסטרקטי שרק יפתור בעיות ספציפיות, ובשאר המקרים יתן לפעולות לרוץ כפי שירצו.
- רובוסטיות - נעילה לפני ואחרי כתיבה - עלות I/O גדולה כאשר יש הרבה תהליכים.

פתרונות אפשריים

- עבודה בלי זיכרון משותף.
- קבלת עזרה מהקומפיילר.
- טרנזאקציות אטומיות.

הגדרה 4.16 טרנזאקציה היא אוסף פעולות המתבצעות ביחד באופן אטומי (על מספר אובייקטים). אם אוסף הפעולות הללו לא מתבצע אטומית, הטרנזאציה נכשלת ומבצעת abort ויכולה לנסות מחדש, אבל אף תהליך אחר לא יראה מצב ביניים זמני. הפעולות בטרנזאקציה לא יכולות להיות אי הפיכות.

טראנזאקציות מאפשרות מקבילות:
אם ניגשות לאובייקטים שונים, לא מפריעות זו לזו. אם טרנזאקציות מקבילות ניגשות לאותם משתנים, תתכן הפרעה שתגרום ל-abort.

4.6.1 מימוש אופטימי מול פסימי

השיטה הפסימית - הטרנזאקציה נועל כל אובייקט שהיא ניגשת אליו, ומשחררת מנעולים רק בסוף. יש חלוקה בין מנעולים לקריאה ולכתיבה (ע"ע בעיית הקוראים-כותבים). כאן יתכן דדלוק - המערכת מגלה אותו ועושה abort.
השיטה האופטימית - הטרנזאקציה זוכרת את הערך הקודם של כל אובייקט שהיא ניגשת אליו, מבצעת שינויים על עותק פרטי, ובסיום בודקת אם איזשהוא אובייקט השתנה בינתיים. אם כן - מבצעת abort, אחרת מסיימת.
היום בדר"כ משתמשים בשיטה האופטימית.

4.6.2 מוניטורים לעומת טרנזאקציות

מוניטורים	טרנזאקציות
פעולה אטומית על אובייקט מסויים	פעולה אטומית כל מספר אובייקטים
המימוש נועל את האובייקט	המימוש לא חייב לנעול
תמיד מצליח	ייתכן abort לאחר ביצוע חלקי
לא כדאי לעשות פעולות I/O - יחסום אחרים	אסור לעשות פעולות I/O - לא יוכלו להתבטל

4.6.3 סיכום

יתרונות

- תכנות פשוט, פחות מועד לטעויות ממנעולים.
- מאפשר מקבילות.

חסרונות

- ביצוע ספקולטיבי:
  - abort מבזבז משאבים.
  - יתכן livelock.
  - בעייתיות עם פעולות לא הפיכות, למשל I/O.
- תמיכה מוגבלת:
  - תקורה כבוהה במימוש בתוכנה.
  - תמיכה בחומרה חלקית\מוגבלת מאוד (כיום).

4.7 ניהול משאבים ודדלוק

במערכת מרובת תהליכים, סך כל המשאבים במערכת בדר"כ לא יספיקו לכל הדרישות של כל התהליכים הנמצאים במערכת בזמן נתון. על כן, התהליכים מתחרים זה בזה על המשאבים. בעיה קלאסית - בעיית הפילוסופים הרעבים.

הגדרה 4.17 קבוצת תהליכים נמצאת במצב דדלוק אם כל תהליך בקבוצה מחכה לאירוע שיכול להגרם רק ע"י תהליך אחר בקבוצה.

האירוע שבדר"כ מחכים לו הוא שחרור משאב.

4.7.1 תנאים הכרחיים להווצרות דדלוק

- mutual exclusion - יש משאבים שאינם ניתנים לחלוקה.
- hold and wait - קיים לפחות תהליך אחד המזיק במשאב וממתין למשאב נוסף.
- non-preemptive allocation - משאב שהוקצה משתחרר רק כאשר השימוש בו מסתיים.
- circular wait - קיימת שרשרת המתנה המכילה לפחות 2 תהליכים, כשכל תהליך מחכה למשאב התפוס ע"י התהליך הבא בשרשרת.

אם יש מופע יחיד של כל משאב, התנאים גם מספיקים - אחרת לא.

4.7.2 פתרונות

- מניעת אחד התנאים ההכרחיים כך שלא יתכן דדלוק על פי תכנון המערכת.
  - מניעת hold and wait:
    - מבקשים את כל המשאבים בבת אחת, אם לא הכל זמין, ממתינים. אבל, זה אומר שמשאבים מוחזקים יותר זמן ממה שצריך.
    - משחררים משאבים שכבר מוחזקים אם לא מקבלים משאבים נוספים שמבקשים. אבל, אולי כשנבקש בעתיד המשאב ששוחרר כבר יהיה מוחזק ע"י תהליך אחר.
  - מניעת מעגלים - מגדירים סדר על המשאבים, ומבקשים אותם בסדר קבוע.
- גילוי והתאוששות - מאפשרים דדלוקים, אבל במקרה שקורה משתמשים במנגנון שמגלה את הדדלוק ומשחרר.
  - מערכת ההפעלה יכולה לתחזק גרף בקשות בתור מבני נתונים. על מנת לאתר דדלוק, נבצע חיפוש בגרף הבקשות וההקצאות.
  - אפשרות נוספת - שימוש ב־watchdog - מנגנון חיצוני למערכת הבודק את התנהגותה.
  - במקרה וגילינו דדלוק - אפשר להרוג תהליך אחד ולהמשיך, או לעשות restart.
- אפשרות נוספת - התעלמות.

מה עושים בחיים?

- מונעים hold and wait.
- לגבי משאבי סנכרון - משתדלים למנוע circular wait - מגדירים סדר על מנעולים הקשורים זה לזה, ותופסים מנעולים בסדר הזה תמיד.
- במערכות מבוססות מנעולים, מריצים מידי פעם גילוי והתאוששות.
- לא מונעים דדלוקים באופן מוחלט.

5 ניהול זיכרון

הזיכרון הוא משאב חשוב. כאשר המעבד מריץ תוכנית, הוא ניגש לזיכרון לצורך ביצוע פעולות מיליארדי פעמים בשניה. על כן, גישה מהירה לזיכרון קריטית לביצועים.

5.1 הגדרת הבעיה והקצאה רציפה

החומרה ומ"ה מסתירות מאיתנו את ההיררכיה ומספקות אבסטרקציה של הזיכרון - מרחב כתובות וירטואלי.
לכל תהליך יש מרחב כתובות וירטואלי משלו - כתובת 1000 של תהליך אחד מכילה מידע שונה מכתובת 1000 של תהליך אחר - והמידע ממופה למקום אחר בזיכרון הפיזי.
ראינו כי מרחב הכתובות של תהליך מוחלק לאיזורים לפי הפונקציונליות שלהם:

- טקסט - הקוד, או הוראות המכונה, של האפליקציה המבוצעת, כפי שנוצר על ידי הקומפיילד.
- data - בדר"כ מכיל מבני נתונים שהוגדרו מראש, לעיתים לפני שהביצוע של התוכנית מתחיל. שוב, נוצר על ידי הקומפיילר.
- heap - אזור להקצאה דינמית של זיכרון.
- stack - אזור לחישובים של פונקציות שהתוכנית קוראת להם. אזור זה גדל במהלך ריצה לפי הצורך.

בין ה־heap ל־stack יש מקום פנוי להגדלה הדינמית.
מערכת ההפעלה ממפה את מרחבי הכתובות (הוירטואלים) של התהליכים לזכרון הפיזי. במיפוי זה נרצה לדון בפרק זה.
מה ההבדל בין הכתובות הוירטואליות לכתובות הפיזיות?
אם שני תהליכים מריצים את אותה התוכנית, אזי לכל משתנה גלובלי foo, הכתובת הוירטואלית שלו $\&foo$ היא זהה בשני התהליכים, למרות שיתכן ויהיו שמורים בו ערכים שונים.
עם זאת, אם מריצים תוכניות שונים, כתובת וירטואלית של תהליך אחד תכיל מידע שונה מאשר אותה כתובת וירטואלית בתהליך אחר.
מערכת ניהול הזיכרון תפקידה מימוש האבסטרקציה של מרחב זיכרון עבור כל התהליכים, מעל RAM ודיסק, באופן יעיל.
מה צריך כדי לממש את האבסטרקציה?

- מיפוי כתובות לוגיות לפיזיות.
- הגנה על זיכרון של תהליך מפני גישות של תהליכים אחרים.
- הקצאה של מקום בזיכרון לתהליכים.

ניהול זיכרון מחייב שיתוף פעולה בין מערכת ההפעלה לחומרה:

1. מיפוי והגנה בכל גישה לזיכרון חייבת להתבצע בחומרה.
2. הקצאה ועדכון טבלאות\רגיסטרים עבור מיפוי והגנה מתבצעים ע"י מערכת ההפעלה - כלומר בתוכנה.

מיפוי והגנה בצד של החומרה נעשים ע"י רכיב חומר שנקרא ה־MMU (בתוך המעבד). מערכת ההפעלה לא מתערבת בגישות רגילות לזיכרון (אפשר לחשוב על מערכת ההפעלה במקרה זה כ"תומכת לחימה".

הגדרה 5.1 external fragmantation - פרגמנטציה חיצונית מתרחשת כאשר יש מספיק מקום לא מוקצה בזיכרון עבור התהליך המבוקש, אך לא ניתן לטעון אותו כי הזיכרון מפוצל (לא מאורגן באופן רציף).

הגדרה 5.2 internal fragmantation - פרגמנטציה פנימית מתרחשת כאשר יש שטחים לא מנוצלים בתוך המחיצות.

5.1.1 הקצאה רציפה

- כל מרחב הכתובות (הזיכרון הוירטואלי) של תהליך יושב באופן רציף בזיכרון הפיזי.
- כל תהליך במערכת נמצא בשלמותו בזיכרון הראשי. אם אין מקום בזיכרון - מבצעים $swap$: הורדה של תהליכים שלמים לדיסק.

לא משתמשים בשיטה זו היום.
בכל זאת, כיצד ממשים?
מיפוי והגנה בצד של החומרה נעשים ע"י רכיב חומר שנקרא ה־MMU (בתוך המעבד) - base שמכיל את הערך של הכתובת הפיזית בקטנה ביותר, ו־limit המכיל את הטווח של הכתובות הלוגיות - כל כתובת לוגית חייבת להיות קטנה מהערך ברגיסטר זה.
מערכת ההפעלה לא מתערבת בגישות רגילות לזיכרון. רכיב זה משתמש ברגיסטרים ובטבלאות שמערכת ההפעלה מעדכנת.
כשתהליך מזומן לריצה, מנהל הזיכרון מחפש את ה"חור" גדול מגודל התהליך, ומקצה בו זיכרון בגודל הנדרש. בסיום, הזיכרון משוחרר, ומתאחד עם חור שכן (אם יש כזה). במקרה כזה על מערכת ההפעלה "למלא חורים" - מכיוון וכל תוכנה צריכה בלוק רציף, וכשתוכנות מסיימות לרוץ נקבל "חורים" לאו דווקא רציפים בזיכרון.
ישנם אלגוריתמים שדואגים "למלא חורים", בוחרים איזה אלגוריתם מתאים לפי העבודה של מערכת ההפעלה:

- first-fit - מקצה את החור הראשון שגדול מספיק.
- next-fit - כמו הקודם, אבל מקצה את החור הראשון מההקצאה האחרונה שגדול מספיק.
- best fit - כשפותחים תוכנית, מחפשים את החור הכי קטן שיכול להכיל את הזיכרון של התוכנה - טוב במקרה שרב התוכנות שרצות צריכות בערך את אותו הזיכרון. אלגוריתם זה גורם לכך שהחור שישאר הוא הקטן ביותר.
- worst-fit - מחפש את החור הכי גדול - טוב במקרה שיש תוכנות שצריכות זיכרון גדול ותוכנות שצריכות מעט זיכרון.

יתרונות:

- פשוט, מסתפק בתמיכה פשוטה בחומרה (עבור מיפוי והגנה).

חסרונות:

- פרגמנטציה גבוהה (שליש).
- קשה לתמוך בגודל משתנה של תהליכים (תהליכים גדלים בזמן ריצה).
- כל הזיכרון של תהליך חייב להיות בזיכרון - לכן קשה לתמוך בתהליכים גדולים, וכמו כן מגביל את מספר התהליכים בזיכרון (במצב ready).
- swap יקר (מעבירים תהליך שלם לדיסק) - אם האיזור המוקצה לכך מלא, הורגים תהליך.

הפתרון:

5.2 *Paging*

זוהי השיטה בה משתמשים היום במערכות הפעלה מודרניות.
מחלקים את הזיכרון הפיזי לבלוקים בגודל קבוע בשם frames - מסגרות (הגודל הוא חזקה של 2 על מנת להקל על המיפוי), ואת הזיכרון הווירטואלי ל- pages - דפים, באותו גודל שווה. שומרים באופן כלשהוא את המסגרות הפנויות.
לא בהכרח כל הדפים של תהליך רץ נמצאים בזיכרון - נעשה שימוש ב-demand paging: הבאת דפים מהדיסק לפי צורך.
הכתובת הוירטואלית נחלקת לשניים: מספר הדף, וכתובת בדף (offset).
טבלת דפים לכל תהליך ממפה מספר דף למספר מסגרת (מספר הדף משמש כאינדקס לטבלה), מעודכנת ע"י מערכת ההפעלה. התרגום עצמו נעשה בחומרה.
עבור קוד משותף, עותק אחד לקריאה בלבד משותף בין תהליכים. כל טבלת דפים ממופה לאותו עותק פיזי של הקוד המשותף.
עבור קוד פרטי ומידע - כל תהליך שומר עותק נפרד של הקוד והמידע.
כמו קודם, צריך רגיסטרים שישמרו את המיקום של טבלת הפייג'ים - page-table base register, ואת גודל טבלת הפייג'ים (כדי שלא ניגש למשהו שלא הוקצא) - page-table length register.
demand paging עובד תחת הנחה של לוקליות: בזמנים סמוכים תוכניות ניגשות לאותם הדפים. היום יש מקומות שמשתמשים ב-paging ללא demand paging - בשרתים עם הרבה זיכרון ודרישות ביצועים גבוהות, ובמכשירים ניידים חסרי דיסק (טלפונים חכמים).

יתרונות:

- גמישות בהקצעת זיכרון - לא חייב להיות רציף.
  - מונע פרגמנטציה חיצונית\פנימית (חוץ מאשר בדף האחרון).
  - יכולת גידול נוחה.
- עם demand paging אין תלות בגודל הזיכרון הפיזי.
  - תוכניות גדולות מגודל הזיכרון יכולות לרוץ.
  - סך הזיכרונות של כל התוכניות ב-ready queue יכול להיות גדול מהזיכרון הפיזי.
  - הגדלת מספר התהליכים בזיכרון - degree of multiprogramming.

5.2.1 טבלאות *paging*

כאמור, לכל תהליך יש טבלת דפים משלו. בטבלת הדפים יש כניסה לכל דף - Page Table Entry (PTE) המכילה:

- מיפוי למסגרת בזיכרון.
- ביט valid שאומר אם הדף מוקצה ונמצא בזיכרון. אם כן, מתבצע תרגום. אחרת - page fault (סוג של פסיקה) - בקשה ממ"ה להתערב.
- ביט modified/dirty.
- הרשאות כתיבה\קריאה.
- מידע לגבי שימוש בדף עבור מ"ה (לצורך החלפת דפים).
- ביט copy-on-write.

הטבלאות מאוד גדולות, מה שגורם לבעיות:

- לא ניתן לשמור את הטבלאות בחומרה ייעודית במלואן, שכן במקרה זה המיפוי יהיה מאוד איטי.
- הטבלאות מבזבזות הרבה מקום בזיכרון הראשי.
- בארכיטקטורות של 64 ביט, הבעיה אפילו יותר חריפה.

מה עושים? נטפל תחילה בבעיה הראשונה:

5.2.2 TLB - שימוש בזיכרון מטמון

בהנחה שהלוקליות נשמרת (כפי שאנו יודעים שקורה לרב), בונים זיכרון מטמון של כניסות מטבלת הדפים, שפונים אליו עם שאילתא: "מצא לי את האינדקס הזה בזיכרון". כלומר, אין צורך לחשב פונקציה, אלא שואלים אם הדף נמצא, ומקבלים מיד תשובה. זה נעשה בסייקלים בודדים בחומרה ע"י ה-MMU, ללא התערבות מערכת ההפעלה.

כאשר ה-cpu מבקש לפנות לכתובת מסויימת, מפעילים את הכתובת על זיכרון מטמון, את מה שמוחזר (במידה והוא נמצא) מחברים ל-displacement ואז קיימת כל הכתובת וניתן לגשת לזיכרון המבוקש. אם הדף המבוקש לא נמצא בזיכרון מטמון, ניגשים לטבלה, כפי שראינו בעבר.

אם בגישה ל-TLB לא מוצאים את הכניסה המבוקשת, ה-MMU ניגש לטבלת הדפים (ללא התערבות מערכת ההפעלה), שומר את התשובה ב-TLB, ובמידת הצורך זורק שורה כדי לפנות מקום - במקרה זה מעתיקים dirty bit לטבלת הדפים.

כאשר יש context switch - מסמנים את כל הכניסות ב-invalid, או הוספת שדה ASID עבור TLBs גדולים.

אם רב הגישות נעשות דרך ה-TLB - הביצועים טובים.

לכאורה, אם היו דפים מאוד גדולים, היה אפשר למפות את כל הזיכרון ל-TLB, אבל במקרה הזה היו הרבה רווחים בזיכרון, אז יש כאן tradeoff. מהדוגמא בשקופית 40, קל לראות כי ה"עונש" בגישה ל-TLB והגעה ל-$miss$ (כלומר אם ניגשנו ל-TLB אבל המידע שחיפשנו לא נמצא בו, ויש צורך לגשת לטבלה למציאת הכתובת) הוא קטן מאוד.

כיום יש שימוש במספר רמות של זיכרון מטמון.

5.2.3 Multi-Level/Hierarchical Paging - היררכיות

ככל שהזיכרון גדל, הטבלאות מאוד גדולות ותופסות הרבה מקום מהזיכרון הראשי. בעוד שבעזרת ה-TLB הגישה מהירה יותר, היא לא פותרת את בעיית בזבוז המקום. שיטת ה-Multi-Level Paging היא דרך אחת להתמודד עם בעיה זו.

הרעיון: ליצור מספר רמות של indirection. היום משתמשים ב(עד)-4 רמות במחשבים שונים. הרמה הראשונה חייבת להיות בזיכרון הראשי, השאר יכול להיות בדיסק. הבעיה: גישה זו מגדילה את מספר הגישות לזיכרון הראשי - אם הטבלה לא נמצאת ב-TLB, צריך לייבא אותה, ואז לייבא את המידע הדרוש. יתרון: הקצאת דפים לטבלה פרופורציונלית לגודל התהליך.

אפשר לפתח את הגישה הזו הלאה - ציינו כי שימוש בדפים גדולים לתהליכים שצריכים דפים גדולים עוזרת. מגבילים את מספר סוגי הדפים (בשקופית 45 יש שני סוגים), ואז אפשר לקבל מערך דפים הרבה יותר עשיר ממקודם. הטבלה החיצונית כאמור תמיד בזיכרון, הפנימיות מוקצות לפי הצורך.

5.2.4 Inverted Page Table

פתרון לבזבוז המוגבר ב־64 ביט.
הרעיון: במקום לחלק לדפים אבסטרקטיים ולשמור לינקים, שומרים את המקום האמיתי.
שומרים טבלה יחידה (לא טבלה לכל תהליך) הפוכה בה כניסה לכל מסגרת בזיכרון הפיזי.
כל כניסה שכזו מכילה:

- $ASID$ - מזהה של מרחב הכתובות אליו המסגרת מוקצה.
- p - כתובת וירטואלית של דף במרחב הזה.

יתרון - חסכון במקום. חסרון - חיפוש איטי. איך נשפר את החיפוש? בשיטה הכי יעילה שאנו מכירים - על ידי פונקציית האש לחיפוש כניסה בטבלת הדפים ההפוכה.

5.2.5 Hashed Page Table

שיטה זו נפוצה במערכות בהן גודל כתובת גדול מ־32 ביט.
מצמצמים את מרחב החיפוש (לא מצמצם את גודל הטבלה כמובן) ע"י פונקציית האש:
גישה אחת לטבלת ההאש, גישה שניה לטבלת הדפים. טבלת ההאש תהיה גדולה מטבלת הדפים למניעת התנגשויות.
מספר הדף הוירטואלי עובר דרך פונקציית האש לטבלת דפים. טבלת הדפים מכילה שרשרת של אלמנטים שפונקציית ההאש הקציבה אותם לאותו המיקום. כדי למצוא את הדף המבוקש, מחפשים בשרשרת את מספרו. אם נמצאה התאמה, הפריים הפיזי המתאים מוחזר.

5.2.6 סיכום - פתרונות לגודל טבלאות ה־paging

- שימוש ב־TLB למיפוי מהיר רב הזמן.
- שימוש בטבלאות בכמה רמות לחסכון בשטח הטבלה.
- שימוש בטבלה הפוכה כדי להקטין את הטבלה.

בעיה עם paging:
חלוקה לדפים שרירותית - לא תואמת את מבנה התוכנית. למשל, קוד ($text$) ו־data יכולים להיות באותו הדף, ואז בעייתי לקבוע הרשאות לדף.
מה עושים?

5.3 סגמנטציה

סגמנטציה היא סכמת ניהול זיכרון התומכת באופן בו המשתמש רואה את הזיכרון.
מחלקים הזיכרון לסגמנטים - חלקים שונים המתאימים לוגית לתוכנית: תוכנית ראשית, פונקציה, מתודה, אובייקט...
בשונה מ־paging, לסגמנטים גודל משתנה, והם נקבעים ע"י הקומפיילר, בעוד שב־paging החומרה מפצלת את הכתובת ללא ידיעת המשתמש.
בשיטה זו, כתובת לוגית מחולקת לשני חלקים: מספר הסגמנט, והאופסט בתוך הסגמנט.
המיפוי נעשה על ידי טבלת סגמנטים בחומרה- טבלה זו ממפה את הכתובת הפיזית לשני חלקים - כל שורה בטבלה מכילה:

- base - הכתובת הפיזית הראשונה בה הסגמנט יושב.
- limit - האורך של הסגמנט.

כמו כן נעזר בשני רגיסטרים לבדיקת חוקיות - segment-table base register מצביע למיקום של טבלת הסגמנט בזיכרון, ו־segment-table length register המציין כמה סגמנטים מנוצלים על ידי תוכנית. מספר סגמנט s הוא חוקי אם $s<\text{STLR}$.
סגמנטיישן פולט - מנסים לגשת למיקום שלא קיים בתוך הסגמנט (גדול מידי).
ניתן להגדיר הרשאות לכל סגמנט. אזי כל גישה למיקום בסגמנט תקושר לביט אישור, וכמו כן ניתן להגדיר לסגמנטים שונים הרשאות שונות - קריאה\כתיבה\ביצוע.
בעיות:

- כל פעם שניגשים צריך לבדוק שלא חורגים.
- היכולת לגשת לזיכרון משתנה מריצה לריצה. מה זאת אומרת? כתובת בזיכרון היא (נגיד) 16 ביטים, ונגיד 6 ביטים מייצגים את מספר הסגמנט, ו־10 את המיקום. כתוצאה מה"חירור" בזיכרון ששונה מריצה לריצה, בריצה הבאה נקבל כמות אחרת של זיכרון, וכך מרחב הכתובות משתנה. היינו רוצים שלתוכנות לא יהיה אכפת מהנושאים האלו, לקבל אבסטרקציה כמה שיותר גדולה.

אלגוריתם 3 האלגוריתם האופטימלי OPT להחלפת דפים
החלף את הדף שישתמשו בו הכי רחוק בעתיד.

5.3.1 שילוב של סגמנטציה ו־Paging

על מנת למנוע פרגמנטציה חיצונית, ועדיין לשמר סגמנטים לוגיים, ניתן לחלק סגמנטים לדפים ולהשתמש ב־paging.
בשיטה זו, לכל תהליך טבלת סגמנטים, ולכל סגמנט טבלת דפים. אז מבנה כתובת אפשרי יכיל שלושה חלקים - חלק למספר הסגמנט, חלק למספר הדף, וחלק ל־offset.
במקרה זה לא תהיה פרגמנטציה חיצונית, אבל תהיה פרגמנטציה פנימית מכיוון וההשמות הן מכפלות של גדלי הדפים.

5.4 זיכרון וירטואלי

5.4.1 עוד על demand paging

תזכורת: לא כל הדפים נמצאים בזיכרון במשך ריצת התהליך. כשתוכנית מתחילה לרוץ, מ"ה מנחשת אילו דפים יהיו בשימוש וטוענת רק אותם לזיכרון. על מנת להבדיל בין דפים בזיכרון לאילו שאינם בזיכרון (או לא מוקצים), לכל דף יש valid bit בטבלת הדפים (מציין אם הוקצתה לדף מסגרת), ביט זה מעודכן על ידי מערכת ההפעלה כאשר הדף ממופה.
הבאת\הקצאת דפים נוספים לפי דרישה (on demand). כשתהליך מנסה לגשת לדף שאינו קיים בזיכרון - החומרה מייצרת פסיקה מסוג page fault, ומערבת את מערכת ההפעלה.
אם הדף לא קיים בדיסק - מחזירים שגיאה למשתמש.
אחרת, מערכת ההפעלה:

- מוצאת מסגרת בזיכרון לדף המבוקש.
- מבצעת פעולה I/O להביא את הדף מהדיסק.
- עד שהדף יגיע, התהליך במצב $wait$, ומבוצע context-switch.

כשהדף מגיע מהדיסק, מערכת ההפעלה מעדכנת את טבלת הדפים ואת ה־valid bit, והתהליך עובר למצב ready. כאשר התהליך מזומן שוב לריצה, פעולתו ממשיכה כאילו הדף לא היה חסר מלכתחילה.
מערכת ההפעלה בדרך כלל מחזיקה מאגר - pool של מסגרות פנויות - ואז כאשר נדרשת מסגרת, אין צורך להמתין לסיום כתיבה לדיסק של הדף שנזרק.
בתוך מסגרת נתונה מחליפים את הדף שנמצא בה בהתאם לדינמיקה של התהליך. אפשר לזרוק דפים כשצריך, וכן אפשר לשמור מספר מסויים של מסגרות פנויות. הפעולה השניה "הגיונית" כי אם המידע השתנה, לפני ש"זורקים" את הזיכרון ממסגרת צריך לשמור אותו לדיסק כדי לא לאבד את המידע, פעולה שלוקחת הרבה זמן. כמו כן, מידי פעם כותבים לדיסק עם modified bit (מבלי לזרוק) ומאפסים את ה־bit.

5.4.2 אלגוריתם החלפת דפים

אלגוריתם החלפת הדפים (Page Replacement Algorithm) הוא אלגוריתם שמריצה מערכת ההפעלה, המחליט איזה דף לזרוק כשצריך למצוא מקום לדף חדש. הוא צריך להיות מהיר (אם לוקח לו הרבה זמן זה לא יעזור לנו), ולמצוא דרך לפנות מקום בצורה יעילה לדף חדש.
אם הדף שבחרנו לזרוק השתנה מאז הכתיבה האחרונה, יש לכתוב אותו לפני הבאת הדף החדש - הוא מכפיל את ה־pft - זמין שירות ל־pf. איך יודעים אם הדף השתנה? לפי ה־modified bit בטבלת הדפים. מי מדליק את ה־modified bit? החומרה, לא מערכת ההפעלה! יש גם ביט שמסמן האם ניגשנו אל הדף, נגיע אליו בהמשך.
כיצד ניתן להשוות ביצועים של אלגוריתמים שונים? בעזרת סימולציות, כאשר המדד העיקרי לביצועים הוא קצב ה־page faults.
כ־baseline בעת הערכת ביצועי אלגוריתם כלשהוא, אנו משווים אותו לביצועים של האלגוריתם האופטימלי, המבטיח את הביצועים הטובים ביותר עבור כל קלט. במקרה של אלגוריתם החלפת דפים, האלגוריתם האופטימלי לו אנו משווים יהיה אלגוריתם offline - "אלגוריתם שיודע לחזות את העתיד" - הוא מקבל כקלט את סדרת הבקשות (בעוד שאלגוריתם אמיתי הוא אלגוריתם online - מקבל את סדרת הבקשות במהלך הריצה, ולא יודע מה יבקשו ממנו בעתיד). הוא מבטיח ביצועים טובים ביותר עבור כל קלט. הוא תיאורטי ולא ניתן למימוש, אך משמש כ"נייר לקמוס" לבדיקה איך האלגוריתם שלנו פועל יחסית אליו.

למרות שאנחנו לא יודעים "לחזות את העתיד" לאלגוריתם אמיתי יש עדיין מידע מסויים אותו הוא מקבל ממערכת ההפעלה:

- מערכת ההפעלה יודעת אילו דפים בזיכרון, ואילו בדיסק - מנהלת טבלאות דפים.
- מערכת ההפעלה יכולה לדעת לגבי כל דף בזיכרון מתי הוא נטען לזיכרון.

עם זאת, היא לא יודע מה קורה לכל דף בזיכרון בין page faults.
נביט במספר אלגוריתמים אפשריים:

5.4.3 אלגוריתם $FIFO$

הקריטריון: בוחרים בדף הוותיק ביותר להחלפה. ממומש בעזרת מחסנית.
האינטואיציה אומרת לנו כי הוספת מסגרות אמורה להקטין את מספר ההחטאות. אולם מכיוון והאלגוריתם לא יודע מה היה קודם, כאשר מגדילים את מספר המסגרות, יש סדרות גידות עבורן הוא ירוץ פחות טוב. זוהי אנומליה לא טבעית - Belady's Anomaly. לאלגוריתם זה יש כמה כאלה. הוא כאמור כלל לא מוצלח (למרות שקל למימוש ע"י תור FIFO).

5.4.4 אלגוריתם Least Recently Used

ניבוי העתיד בעזרת העבר הקרוב.
הקריטריון: החלף את הדף שהיה בשימוש בזמן הרחוק ביותר.
הוא יעיל, אך יקר למימוש - צריך לזכור זמן גישה אחרון לכל דף, ולתחזק מבנה נתונים המתעדכן בכל גישה לזיכרון. מערכת ההפעלה לא יכולה לתחזק מבנה כזה, ולכן צריך שהחומרה תטפל בזה. למנגנון המתוחזק על ידי החומרה צריך שהלוגיקה תהיה פשוטה יחסית.
על כן, לא משתמשים ב־LRU טהור.

5.4.5 Clock: Second-Chance Algorithm

לכל דף בטבלת בדפים מוצמד ביט גישה - רפרנס R - כל גישה לדף מדליקה את הביט (ובפרט טעינה של הדף לזיכרון מדליקה את הביט). ההדלקה מבוצעת ע"י החומרה.
יש תור FIFO של דפים (מימוש יעיל: תור מעגלי). כשאלגוריתם FIFO אמור לזרוק דף, בודקים את הביט רפרנס, אם הוא מודלק מדלגים אליו (נותנים לו second chance), אך מזיזים אותו לסוף התור, ומאפסים את הביט (הביט מאופס ע"י הדיימון ת'רד - מערכת ההפעלה).
אם ניגשו לכל הדפים, האלגוריתם יעשה סיבוב שלם, ונקבל FIFO.
מה החולשה של מנגנון זה? בזיכרון מאוד גדול לוקח הרבה זמן לגמור את המעגל, לכן בריצה השניה יקח הרבה זמן למצוא ביט 0 (כל הדפים יטענו מחדש עד שנגיע אליהם שוב).

5.4.6 2-Handed Clock

פתרון לחולשה שראינו בפתרון הקודם (במערכות Unix ו־Linux): הגדרת חלון זמן בו נבדוק את הדפים.
נקבל שתי "ידיים" לשעון: המחוג הקידמי מאפס את הביט, המחוג האחורי בודק אותו. המרחק בין שני המחוגים מהווה את חלון הזמן בו בודקים שימוש - כך אם באמת מדובר בקובץ בשימוש תדיר, הביט יהיה דלוק בבדיקת המחוג האחורי.
זהו פתרון די חדש, שהגיע כאשר הזיכרון גדל מאוד.

5.4.7 Enhanced Second-Chance

כעת יש לנו כמה ביטים בטבלת הדפים (ועוד שלא נדבר עליהם כעת):

- Valid - האם הדף בזיכרון. מעודכן ע"י מ"ה בלבד.
- Modified/dirty - האם הדף השתנה מאז כתיבתו האחרונה לדיסק.
  - מאותחל ל־0 כאשר הדף נטען.
  - מודלק בכל כתיבה לדף אוטומטית על ידי החומרה.
  - מכובה כאשר הדף נכתב לדיסק אוטומטית על ידי החומרה.
- Referenced - האם ניגשו לדף לאחרונה. מודלק בכל גישה ע"י החומרה, מכובה כאשר המחוג עובר אותו (מאופס ע"י הדיימון ת'רד - מערכת ההפעלה).

נסתכל על קומבינציה של ביטים (R, M) במקום על ביט אחד - יש ארבע קומבינציות אפשריות:

- $(0, 0)$ - כלומר לא ניגשו אליו לאחרונה ולא שינו אותו מאז הכתיבה האחרונה לדיסק. קורבן טוב ביותר, בעדיפות ראשונה לזריקה.
- $(0, 1)$ - פחות טוב מהראשון - לא ניגשו אליו לאחרונה, אבל יש לבצע כתיבה לפני ש"זורקים" אותו. במקרה זה מוציאים בקשת I/O לכתיבת הדף, וממשיכים לחפש מסגרת לשימוש מיידי.
- $(1, 0)$ - ניגשו אליו לאחרונה, לכן קרוב לוודאי שיהיה בשימוש בקרוב.
- $(1, 1)$ - כמו הקודם, רק שגם ידרוש כתיבה, ועל כן דף עם צירוף כזה הוא בעל העדיפות הנמוכה יותר לזריקה.

האלגוריתם אומר: החלף את הדף הראשון במחלקת העדיפות הנמוכה ביותר. הוא יכול לדרוש יותר מסיבוב אחד של סריקה של התור המעגלי. אם זה דרוש - זה לא עובד טוב...

5.4.8 סיכום - אלגוריתמים להחלפת דפים

- OPT - כלי תיאורטי כבסיס להשוואה עבור אלגוריתמי החלפת דפים.
- FIFO - הכי פשוט לשימוש, אפשר לממש ללא עזרה מהחומרה.
- LRU - מצויין עבור workload טיפוסי, אך קשה למימוש (צריך הרבה מהחומרה).
- Clock: 2nd chance ודומיו - קרוב ל-LRU אך פשוט למימוש.

5.4.9 תפוקה ו-demand paging

תזכורת: במערכות demand paging תהליכים אינם נמצאים בשלמותם בזיכרון - מאפשר להגדיל את מספר התהליכים בזיכרון. עם זאת, אי אפשר להוסיף תהליכים עד אינסוף, כי כל תהליך צריך כמות דפים מינימלית כלשהיא. לכן בשלב מסויים הניצולת תרד בצורה דראסטית - אז מתחיל thrashing: כמעט כל פעולה דורשת גישה לזיכרון.

הגדרה 5.3 מערכת נמצאת במצב thrashing אם היא מבלה יותר זמן בניהול (context switch, paging) מאשר בעבודה (הרצת תהליכים).

הגדרה 5.4 תהליך נמצא במצב thrashing אם הוא מבלה זמן רב יותר ב-context switch מאשר בביצוע (ריצה).

ה"אומנות": למצוא כמה תהליכים אפשר להריץ מבלי להגיע ל-thrashing.
working set - לאילו דפים תהליך פונה בזמן מסויים. זה משתנה מתהליך לתהליך. אם נלמד את ה-working sets נדע כמה תהליכים נוכל להריץ בצורה טובה (מבלי להגיע ל-thrashing) - בדרך כלל תוכנית עוברת מ-working set אחד למשנהו. בגלל העלות הגבוהה של page fault, חשוב להסתכל על טווח זמן גדול.
מערכת מגיעה למצב זה אם:

- בתוכנה אין לוקאליות, כלומר, היא פונה כל הזמן למקום שלא השתמשה בה הרבה זמן, ולכן צריך לטעון זיכרון שונה שוב ושוב.
- לא ניתן להחזיק את הזיכרון שהתוכנה צריכה (working set) כדי לפעול בזיכרון הפנוי.

הפתרון:

- הוספת זיכרון.
- הרצת פחות תוכנות.
- להשתמש בתוכנות חסכניות יותר בזיכרון.

עוד פתרון - swapping - אפשר להעביר את תהליך באופן זמני מחוץ לזיכרון לאיזור בשם backing store (גדול מספיק בשביל להכיל עותקים של כל ה-memory images עבור כל המשתמשים) ולשנות את מצבו ל-swapped, ואז שאר התהליכים יכולים להמשיך לרוץ. ה-backing store צריך להיות גדול מספיק בכדי להכיל עותקים עבור כל המשתמשים.

6 מערכות קבצים - File Systems

ההבדלים בין הדיסק והזיכרון הראשי:

דיסק (זכרון משני)	RAM (זיכרון ראשי)
גדול	קטן
זול	יקר
עקבי - המידע נשאר עד שניגשים אליו בפעם הבאה	נדיף - כשלא מקבל מתח - המידע מתנדף
איטי	מהיר
אין גישה ישירה למידע מ-cpu (מכיוון ואיטי)	נגיש ישירות ע"י ה-cpu (ע"י ממשק של זכרון וירטואלי)

למה צריך מערכת קבצים?

- שמירת תוכניות ומידע כשתהליך מסתיים או נופל.
- שיתוף מידע בין תהליכים שונים.
- ניתוק המגבלות של המידע במחשב (גודל + מיפוי) מהמידע עצמו.

אבל כבר דיברנו על ניהול זיכרון, מה ההבדל? ובכן, אנחנו נראה כי אפשר להשתמש בחלק מהשיטות שהשתמשנו בהם בניהול זיכרון. אולם כאן, תפקיד מערכת ההפעלה הוא שונה:

בניהול זיכרון המיפוי וההגנה נעשו ע"י החומרה, ותפקיד מערכת ההפעלה היה רק הקצאה ועדכון טבלאות לתמיכה בפעולת החומרה - מערכת ההפעלה לא היתה מעורבת בגישה לזיכרון, אלא היתה מעין "תומך לחימה".

לעומת זאת, כאן אנחנו נראה כי מערכת ההפעלה מספקת אבסטרקציה, וכן גם מנהלת את הגישה לקבצים - זהו הבדל גדול שחשוב להבין (והוא מה שגם מרשה להשתמש ב-LRU ב-buffer cache, בעוד שאנחנו לא יכולים לעשות זאת עבור paging).

דרישות

- זיכרון לא נדיף.
- מקום אחסון גדול מאוד.
- הגנה על מערכת הקבצים - בקרת גישה.
- בקרת מקביליות על גישה למידע (ניגע רק מעט בנושא זה).

6.1 האבסטרקציה

מערכת קבצים היא אבסטרקציה שמערכת ההפעלה מספקת. האבסטרקציות העיקריות הן קובץ, ומדריך\מחיצה.

הגדרה 6.1 קובץ הוא יחידת מידע לוגית שיש לה שם, והיא לא נדיפה - היא נשמרת לאורך זמן ממושך יותר מזמן ריצת התוכנית שיצרה אותה.

תהליכים יכולים לגשת לקובץ מאוחר יותר, או בו זמנית.

קובץ הוא מבנה נתונים מופשט ADT. המשתמש אינו יודע איך והיכן הוא נשמר. לקובץ יש:

- שם הניתן על ידי המשתמש.
- תוכן\מידע\data.
- תכונות - attributes (תלוי מערכת הפעלה).
- פעולות - operations.

קובץ יכול להיות מוגדר בדרכים שונים:

- חסר מבנה - רצף בתים. למשל, קובץ רגיל ב-Unix. הגישה למידע היא עפ"י היסט (offset) מתחילת הקובץ.
- מובנה - אוסף של רשומות. מאפשר גישה לפי מפתח או מפתחות.

כיצד נבדיל בין השניים? צריך להיות רשום לנו איפשהו מהו המבנה. מידע לגבי הקובץ נקרא meta-data, וגם הוא נשמר בדיסק. מכיל למשל את:

- סוג הקובץ.
- מיקום - שם ההתקן (device) ומיקום ה-data על ההתקן (כתובות בלוקים).
- גודל.
- בעלים.
- הרשאות.
- זמני יצירה, גישה, שינוי אחרון.

איך מזהים את סוג הקובץ ולאיזה אפליקציה הוא מתאים?

אפשר היה לכתוב את כל המידע באופן מסודר, אבל אז כל גישה לקובץ היתה לוקחת הרבה זמן.

במהלך השנים התגבשה הגישה הבאה: לקובץ יש תוספת סטנדרטית (כמו exe ב-*dos*) המאפשרת זיהוי מהיר של ההקשר של הקובץ.

ב-Unix - בכוונה יש תמיכה במעט סוגים - KISS - Keep It Stupidly Simple:

- רגיל (רצף של בתים).

- $FIFO$.
- $directory$.
- symbolic link.

ישנן דרכים נוספות שמ"ה או מערכת הקבצים משתמשת כדי לדעת מאיזה סוג הקובץ - סדרת הבתים בתחילת הקובץ מוגדרת כ־magic number לעזרה בזיהוי.

6.1.1 Memory mapped files

הרעיון: היינו רוצים להסתכל על קובץ כוקטור ענק. שיטה זו ממפה את הקובץ או חלק ממנו לאיזור רציף בזיכרון הוירטואלי - ואז עובדים עליו כוקטור מאוד גדול, ללא קריאות מערכת. דרך זו מפשטת את התכנות וחוסכת תקורות בביצועים.

לא להתבלבל עם memory mapped I/O!

שיטה חלופית לגישה לקבצים - במקום לעשות קריאות מערכת $read, write$ לכל גישה, ממפים את הקובץ לזיכרון בקריאה אחת (mmap), ואז ניגשים לכתובות בוקטור בזיכרון הוירטואלי ומשנים את התוכן, ללא קריאות מערכת.

ניהול הגישות לקובץ - דרך מערכת ניהול הזיכרון - מבוסס על סימון דפים ב־page table כקשורים לאיזור ב־page cache שמחזיק את חלק הקובץ המתאים, אם invalid, אז קישור למקום בדיסק בו הקובץ נמצא.

- יתרונות:
  - היתרון הגדול: המיפוי מאוד יעיל ועל כן הגישה מאוד יעילה לקובץ - חוסכת תקורה של קריאות מערכת.
  - מקל על התכנות.
  - חוסך העתקות בין disk cache ל־user space בזיכרון.
  - חוסך כתיבה לאזור ה־swapped.
- מגבלות -
  - הקובץ חייב להיות מספר אינטגרלי של דפים, אחרת מבזבזים מקום. בעיה שולית היום, כי לרב קבצים מאוד גדולים.
  - אם הקובץ לא נמצא בזיכרון, מ"ה צריכה ליבא את הקובץ מהדיסק.
  - לא ניתן למפות קבצים גדולים מגודל הזיכרון הוירטואלי (אפשר חלקים מהם).

6.1.2 Directory

מדריך למציאת קבצים בהנתן שמם - עוזר להגדיר בצורה סימבולית את המיקום בו אנחנו "חושבים" שהקובץ נמצא. בעבר המבנים היו פשוטים, היום המבנים יותר מסובכים.

בדר"כ המבנה הוא עץ, והוא מגדיר את המיקום של הקובץ. זה מאפשר לשמור קבצים באותו שם אך במקומות שונים, והמיקום מבדיל ביניהם.

השם הוא ה"מסילה" להגיע לקובץ. אפשר להגיע בעזרת כתובת יחסית המוגדרת על פי המיקום הנוכחי, בעזרת הכתובת האבסולוטית. בשתי הדרכים, הקובץ מוגדר באופן יחיד. כלומר, שם הקובץ הוא לא רק הסיפא, אלא כל המסלול.

Link/Unlink ב־Unix יש תמיכה ב־(hard) links השוברים את מבנה העץ.

אסור למשתמש רגיל להוסיף לינק ל־directory.

link(ln) מוסיף כניסה ל־directory עבור קובץ קיים. יש לקובץ attribute המציין כמה שמות שונים הוא קיבל - unlink(rm) מוריד כניסה ב־directory, רק אחרי שמספר הכניסות הוא 0, הקובץ נמחק.

זוהי דרך אחת להגדיר יותר משם\פניה אחת לקובץ.

יש גם symbolic link - גרסא מוחלשת. מגדירים מעין פוינטר - קובץ המכיל בתוכו מיקום של קובץ אחר. אם הקובץ שאליו מצביעים ימחק, הפוינטר לא ימצא את הקובץ, אך מראה המקום עצמו נשמר - זוהי ישות זמנית (זו מחיקה אמיתית, בניגוד ללינק רגיל).

גם על directory מגדירים פעולות מיוחדות - mkdir,rmdir, ועוד.

6.1.3 Access control

יותר מאפליקציה אחת יכולה לגשת לישות קובץ - קבצים יכולים להיות משותפים למספר משתמשים. נרצה לבטא את הרצונות שלנו כיוצרי הקובץ לגבי הגישות, ונרצה כי מ"ה תעזור לנו לממש את הרצונות האלו.

הגדרה 6.2 Acces Control List (ACL) היא רשימה של משתמשים וסוגי גישה המותרים לכל אחד מהם. לכל קובץ\מדריך יש ACL משלו.

כמות התיאורים השונים היא עצומה, ואילו היינו רוצים לממש את כולם, הגישה לקובץ היתה בלתי אפשרית. על כן צריך דרך יותר פשטנית.

במערכות הפעלה שונות יש דרכים שונות - ביוניקס מאפשרים שלוש רמות - user,group,universe.
היתרון - זה מאוד פשוט מבחינת מ"ה. החסרון: אי אפשר לבטא את כל מה שרוצים.
יש מערכות המאפשרות הגדרה של מידע meta נוסף. כאמור זה מאט את הגישה לקובץ.
חלופה: סיסמא לכל קובץ ומדריך. עושים את זה רק לקבצים מוצפנים, בשל התקורה הגבוהה לא הגיוני לעשות את זה לכל קובץ.

6.1.4 Consistency Semantics

חשוב להגדיר מה התוצאה מכך שכמה משתמשים מעדכנים קובץ במקביל.
אם נסתכל על סדרת פעולות של משתמש על קובץ מ-open עד close (סדרה כזו נקראת session) בגישת ה-database, רוצים שהסשן הזה יהיה אטומי. לא נרצה אטומיות לגבי קבצים.
ב- Unix consistency semantics:

- כל כתיבה לקובץ נראית מיידית לכל המשתמשים המחזיקים אותו קובץ פתוח, המימוש ע"י image (עותק) יחיד לכל קובץ. ניתן לכמה משתמשים להחזיק באותו המצביע. הסמנטיקה לא מתקיימות במערכות קבצים המשותפות למספר מחשבים ברשת, לכן צריך להיות מאוד זהירים בהנחות שלנו!
- כדי ליעל את העבודה, מ"ה לפעמים "שומרת" שינויים אצלה ומעדכנת בקובץ הרבה זמן לאחר השינוי.

ב-session semantics:

- כתיבה לא נראית מיידית למשתמשים אחרים המחזיקים את אותו קובץ פתוח.
- ההנחה - מיי שיפתח את הקובץ בעתיד יראה את השינויים. לא מבטיחים הרבה לגבי פעולה בו זמנית.
- ניתן להחזיר מספר images לקובץ בו זמנית.

סמנטיקה זו נתמכת למשל במערכת AFS.

6.2 מימוש מערכות קבצים

הדיסק בנוי מפלטות, כאשר קיימים ראשים קוראים\כותבים ש"מביאים" את המידע (כמו המחט של תקליט). אפשר לגעת במספר פלטות באותו הזמן אבל לא באיזורים שונים. על כן, הדיסק עובד יעיל כאשר ניגשים להרבה מידע באופן רציף. הדרייבר מתכנן לדיסק את הפעולות של הראש - חיפוש המיקום הנכון $(seek)$ ופעולת הזזת הראש למקום הנכון איטית בהרבה יותר מפעולת הבאת המידע עצמה, על כן יש לתכנן את התזוזה שלו בצורה יעילה.
הקובץ עצמו יושב על volume:

הגדרה 6.3 $volume$ הוא יחידה אבסטרקטית שיכול להיות פחות מדיסק, או כמה דיסקים, המחולק לבלוקים. מעתה כשנגיד "דיסק" הכוונה ל-volume (דיסק לוגי).

ההקצאה ממפה את הקובץ או חלק מהקובץ לבלוקים בדיסק.
אם הקובץ אינו מבני, דהיינו רצף בתים, אזי מבחינת הדיסק, קל יותר לפזר את הקובץ על איזורים רציפים.
אם מבני - במבניות יש הנחה כלשהיא לגבי הקובץ, ואז עולה השאלה מהי הדרך היעילה ביותר לבצע את המיפוי.

6.2.1 הקצאה רציפה

הקצאת בלוקים רציפים עבור קובץ. במדרך נשמר רק איפה הקובץ מתחיל ומה אורכו. ב-CD/DVD המידע ממופה כך (כי הכתיבה היא יחידה).
פעם נוהל כך המידע בדיסק, וכל תקופה היו מריצים פעולה שהיתה "מסדרת" את המידע כך שלא יהיה הרבה בזבוז. היום כבר לא נוהגים כך.

- יתרונות:

- שיטה פשוטה.
- seek מאוד מהיר תוך כדי גישה יחידה בזיכרון.
- מינימום תזוזת הראש (קריאה רציפה מהירה).

- חסרונות

- פרגמנטציה (מחיקות עלולות להשאיר חורים בגודל לא מועיל).
- בעיה לתמוך בגדילה של קובץ.
- צריך לנחש גודל מראש (קשה).

6.2.2 $Extent - based$

extent - הקצאה רציפה של בלוקים בדיסק.
לקובץ מוקצים מספר extents שיכולים להיות בגדלים שונים, במדריך שומרים את האקסטנט וגודלו.
יתרון - כך אפשר בצורה דינמית להגדיל את הקובץ, וכן כל עוד מספר האקסטנטים לא גדול, ניתן לנהל בצורה יעילה.

- אקסטנטים גדולים:

- חיפוש יעיל.
- קריאות סדרתיות יעילות.
- הבעיה: פרגמנטציה, גם פנימית לקובץ וגם חיצונית (הפסד מקום בין האקסטנטים).

- אקסטנטים קטנים:

- צריך מבנה נתונים נוסף לניהול המיקומים בהם הקובץ נמצא - איזשהוא b-tree.

נשאל, כמה מקום יש לשמור במדריך למצביעים לאקסטנטים, ואיך לארגן אותם? אם יש הרבה אקסטנטים, שמירת רשימה ארוכה לא יעילה. במקום זה, שומרים את מיקומי וארוכי האקסטנטים במבנה כמו B-tree כדי לתמוך בחיפוש יעיל של offset.

6.2.3 Single-block Allocation

זהו המקרה בו האקסטנט הוא בלוק בודד, בלוקים של קובץ פזורים בצורה כלשהיא על הדיסק. במקרה זה הקריאה הסדרתית פחות יעילה - תזוזות רבות לראש הקורא\כותב.
איך מוצאים את האופסט הרצוי בקובץ? רצים לאורך ה־b-tree - אפשר לעשות בכמה דרכים:

- רשימה מקושרת.
- MS-DOS file allocation table.
- אינדקס:
  - Unix I-nodes.
  - NTFS MFT.

6.2.4 Linked list allocation

קובץ כרשימה מקושרת של בלוקים, כל בלוק מכיל מצביע לבלוק הבא. directory entry מצביע לראשון ולאחרון.
מאוד פשוט, אך לא ישים - התקורה מאוד גבוהה, ולא מאפשר גישה אקראית (seek). פתרון - הוצאת המצביעים מחוץ לבלוקים:

6.2.5 File Allocation Table (FAT)

שיטה שיושמה ב־DOS וחלק מהמשכיו.

דומה ללינקד ליסט, עם הוצאת המצביע מהבלוק לטבלה נפרדת - וזה פותר את הבעיה של הגישה האקראית.

מבנה הנתונים הראשי הוא טבלה משותפת לכל הקבצים השומרת את המצביעים, שורה לכל בלוק בדיסק, הנשמרת באזור מיוחד בתחילת הדיסק. כניסה ב־directory מקשרת בין שם הקובץ לשורה בטבלה שמהווה את הבלוק הראשון בקובץ. שורות המהוות קובץ מחוברות זו לזו ב־linked list. בלוקים שאינם בשימוש מסומנים בטבלה ב־0, סימון מיוחד לסוף קובץ, ולבלוקים "רעים" שאסור להשתמש בהם יש סימון מיוחד.

בעיה: אם הטבלה נהרסת, כל הדיסק הלך, כי לא יודעים מה משוייך למה. עם זאת, מכיוון והחלק הזה נקרא הכי הרבה, זהו האיזור שהכי סביר שיהרס.

איפה הטבלה נשמרת? על הדיסק, כי היא חלק ממנו, אחרת אין לדיסק שימוש. מצד שני, היא צריכה להיות חלק מהזמן בזיכרון כדי שיהיה אפשר ליבא קבצים. אילו היתה נשמרת רק בדיסק, גישה סדרתית לא היתה יעילה. על כן הטבלה נשמרת גם בדיסק וגם בזיכרון (נטענת בזמן boot).

- יתרונות:
  - גמיש, קל לתמוך בגדילה של קבצים.
  - אין פרגמנטציה חיצונית.
  - פשוט ליישום, קל להקצות בלוקים.
  - גישה אקראית מהירה (חיפוש בטבלה קטנה בזיכרון במקום בכל הדיסק).
- חסרונות:
  - טבלה גדולה - מצביע לכל בלוק, לכן בדיסקים גדולים לא משתמשים בשיטה זו.

6.2.6 Indexed allocation

עבור כל קובץ, שומרים בלוק הנקרא index block, השומר מצביעים לבלוקים בקובץ, כלומר, אומר איפה החלקים של הקובץ נמצאים. אז הרבה יותר קל לבצע על האינדקס הזה פעולות כדי למצוא איפה הקובץ נמצא. האינדקס בגודל נתון.

נשים לב כי קבצים גדולים דורשים בלוק גדול. מאוד מוצלח לקבצים גדולים, אך בזבזני לקטנים.

יתרון מול FAT: קל הרבה יותר למצוא חלקים של קובץ.

Indirect indexing - עבור קבצים מאוד גדולים, אפשר להחזיר אינדקס של אינדקסים (כמו שעשינו בזיכרון): שמירת כמה רמות של אינדקסים. בצורה כזו אפשר לשמור מיפוי למספר עצום של בלוקים.

הבעיה: יתכן שלכל בלוק צריך לעבור על כל סדרת האינדקסים הזו.

מצד שני - לא צריך לשמור את כל הסדרה בזיכרון: שומרים רק חלק מהטבלאות, ואם צריך מביאים את המידע של האינדקס הבא מהדיסק.

בכמות קטנה של אינדקסים, בסבירות גבוהה מספקים את הצרכים של אפליקציה בנקודת זמן נתונה.

6.2.7 I-Nodes

השיטה המקובלת היום ב־Unix:

מערכת הקבצים מחולקת לשלושה חלקים - superblock, המנהל את המיפוי של בלוקים (מכיל מידע כגון גודל מערכת הקבצים, רשימת הבלוקים הפנויים, וכו'), i-nodes, ובלוקים של מידע, וסדר זה הוא סדר הופעתם על הדיסק. ישנם חלקים נוספים אופציונליים בתחילת הדיסק - אזור לטעינת מערכת ההפעלה, ואיזור ל־swap.

הקובץ מיוצג על ידי מבנה בשם $inode$ - מכיל את ה־metadata, ומצביעים ישירים לבלוקי נתונים וכן מצביעים למצביעים, וכו'.

ב־unix קלאסי : 13 כניסות, מהן 10 ישירות. מהשלוש שנותרו, 1 לא ישירה, שניה לא ישירה כפולה, ושלישית - לא ישירה משולשת. בלוק מצביעים מכיל N מצביעים, כאשר $N = \frac{block\ size}{pointer\ size}$.

טבלאות עיקריות

- טבלת ה־inodes (כל קובץ מופיע לכל היותר פעם אחת בה).
- טבלת הקבצים הפתוחים - כניסה מוקצת בכל פעם שקובץ נפתח. כל כניסה מכילה פוינטר לטבלת ה־inodes וכן את המיקום בתוך הקובץ. באופן זה יכולות להופיע מספר כניסות המצביעות לאותו ה־inode.
- טבלת ה־fd - נפרדת לכל תהליך. כל כניסה מצביעה לכניסה בטבלת הקבצים הפתוחים. האינדקס הוא ה־fd שהוחזר על ידי הפעולה open.

מציאת בלוק פנוי כדי לתחזק את המערכת, הסופרבלוק מכיל טבלה קצרה של inode פנויים.
כאשר משחררים inode, מיקומו נשמר בטבלה זו, ואלם רק אם יש בה מקום. כאשר הרשימה ריקה, הקרנל מחפש בדיסק ומוצא inodes חדשים להוסיף לרשימה. בשביל שיפור הביצועים, הסופרבלוק מכיל את מספר ה־inodes הפנויים במערכת הקבצים.
כאשר תהליך כותב מידע לקובץ, הקרנל מקצה את הבלוק הפנוי הבא מהרשימה.

מציאת קובץ במדריך יש טבלה הממפה את הקובץ לפי שמו לכניסה בטבלת קבצים שנמצאת על הדיסק. לאחר מציאת הכניסה המתאימה, היא נקראת לזיכרון ומועתקת לתוך רשימת הקבצים הפתוחים של הקרנל. לאחר מכן, נבדקות ההרשאות של המשתמש. במידה והוא מורשה, המשתנה של המשתמש - ה־fd, משתנה כך שיצביע על הכניסה הזו ברשימת הקבצים הפתוחים. כל הפעולות על הקובץ מבוצעות דרך ה־fd.

קריאת קובץ בעת קריאה, בלוק מועבר ל־buffer cache (בשל לוקליות - גם אם לא היינו צריכים את כל הבלוק, אולי נצטרך אותו בעתיד), ולאחר מכן מספר הביטים שבוקשו מועבר לזיכרון המשתמש למיקום המבוקש.

כתיבה לקובץ אם כותבים על מקום קיים - פשוט כותבים למיקום המבוקש. אם יש צורך בהקצאת בלוק חדש, אין צורך בקריאת אחד חדש מהדיסק. במקום זאת, מקצים בלוק חדש ב־buffer cache, מסמנים כי הוא מייצג את מספר הבלוק החדש (לפי רשימת הבלוקים בפנויים), ומעתיקים אליו את המידע הרצוי. לבסוף, הבלוקים ששונו נכתבים שוב לדיסק.

שיקולים בקביעת גודל הבלוק Unix המקורי הותאם לקבצים קטנים. במקרה כזה:

- פחות פרגמנטציה - אין פרגמנטציה חיצונית, יש פנימית בקבצי פוינטרים של קבצים דלילים.
- טבלאות ענקיות, גישה אקראית איטית לקבצים גדולים (צריך לעבור על הרבה רמות של indirection).

על כן הקבצים איתם עובדים מכתיבים את ההקצאות. צורת השימוש משתנה עם הזמן, ואיתה גם הצרכים של המשתמשים - היום למשל הקבצים נוטים להיות גדולים יותר. מערכות Unix מודרניות משתמשות בבלוקים גדולים, הקצאת אקסטנטים של כמה בלוקים רציפים יחדיו.

NTFS 6.2.8

כאן לקבצים יש מבנה, הם לא "סתם רצף של בתים". כל קובץ מורכב מ־attributes, כל אחד מהם רצף בתים נפרד.
יש attributes של meta-data, ויש unnames attributes המכילים data - ההתייחסות לשני הסוגים הינה אחידה.
מבנה הנתונים הראשי בשיטה זו הוא Master File Table (MFT) - טבלה המכילה רשומת מידע לכל קובץ במערכת (מקבילה ל־*inode*). אטריביוטים קטנים נשמרים ב־MFT, וגדולים נשמרים ב־extents בדיסק, עם מצביעים ב־MFT. בשיטה זו מדריך מכיל אטריביוטים מה־MFT של הקבצים בו.
יש תמיכה ב־compression. כמו כן התאוששות מנפילות בעזרת מנגנון טראנזקציות (meta-data בלבד).

6.2.9 ניהול שטח דיסק פנוי

מערכת הקבצים מנהלת free list - רשימת פנויים. בעת יצירת\הגדלת קובץ, יש להקצות לו בבלוקים בדיסק - בוחרים מרשימת הפנויים.
במחיקת קובץ, יש לשחרר את הבלוקים שהוקצו לו כדי שיוכל לשמש לקבצים אחרים בהמשך - מעבירים אותם לרשימת הפנויים, ומוחקים את ה־metadata ולא את הבלוקים עצמם כדי לא לבזבז זמן, ובעת כתיבה מתבצעת דריסה.

6.2.10 מדריכים - *directories*

לכל קובץ במערכת הקבצים נשמר מידע במבנה הנתונים שנקרא File Control Block, המכיל את ה־attributes, ומידע המאפשר גישה לבלוקים\extents של הקובץ. צריך לשמור גיבוי שכן אם נמחק לא נוכל להמשיך לעבוד.
לכל FCB יש מזהה יחיד ביוניקס, שנותן אפשרות לגשת לבלוקים ולקבצים. תפקיד ה־*directory* - למפות כל שם קובץ ל־FCB מתאים.

כיצד ניגשים לקובץ? משתמש ניגש לקובץ דרך המדריך. מ"ה שולפת את ה־FCB הרלוונטי שהמדריך מצביע עליו. לאחר מכן ההרשאות נבדקות, ואם ההרשאות מרשות, ה־FCB מספק מידע על מיקום הבלוקים.
יוניקס שומרת את ה־FCB של כל קובץ פתוח בזיכרון. בעת סגירות הקובץ, משחררת את ה־FCB מהזיכרון. המידע של ה־FCB נשמר בדר"כ במקום קבוע בדיסק, הם לא בתוך המדריכים (שם יש רק פוינטרים ל־FCB).
איך זה באמת קורה?
המדריך הוא למעשה קובץ מיוחד המכיל רשימת זוגות מהצורה $(inode, file-name)$. בפתיחת קובץ - מעבר על קבצי הדירקטורי וה־*inodes* לסירוגין, בכל שלב בודקים הרשאות, וכל מדריך בדרך צריך להיות executable.

הצבעה ל־inode בזיכרון לכל תהליך, כפי שציינו, יש טבלת file descriptors. כל fd מצביע לכניסה בטבלה של קבצים פתוחים המכילה file pointers, כל אחד כזה מצביע לכניסה בטבלת in-core inodes (שם כל inode מופיע לכל היותר פעם אחת).
כל פעולת open יוצרת file pointer פרטי. עם זאת fds שונים יכולים להצביע לאותו ה־file pointer, ע"י dup (אותו התהליך), או fork (תהליכים שונים).

$Mount$ - אתחול מערכת קבצים היסטורית המחשב היה עובד ללא קבצים והיו טוענים טייפ\דיסק כזה או אחר - $mount$, ומאז נשמרה המילה הזו (למרות שהיום לא נטען שום דבר אלא נשמרים קישורים).
כשרוצים להוסיף מערכת קבצים, יש "לשתול" אותה איפשהו במערכת הקבצים בו המשתמש עובד - מקום זה נקרא mount point.
יש מערכת שנותנות לדירקטורי לא להיות עץ, אבל זה פותח את הדלת לבעיות שונות ומשונות. אנו נניח כי אכן מדובר בעץ.

Virtual File System - VFS 6.2.11

אין סיבה להתייחס רק לקבצים של יוניקס או וינדואז, כמשתמש וכמפתח נרצה שקיפות בין הטיפוס של הקבצים ואיך הם שמורים.
לשם כך פיתחו את ה־VFS - זהו ממשק סטנדרטי למערכות קבצים. מתחתיו אפשר "לתלות" מערכות הפעלה שונות קלאסיות, ואז אפשר לגשת לקבצים השמורים בצורות שונות, "שמערכת ההפעלה תשבור את הראש" - אין צורך להעביר את הסוג של המערכת הספציפית, המערכת הוירטואלית מזהה את המערכות הקלאסיות ומתמודדת עם כל אחת בדרך המתאימה.

עקביות בעת התאוששות מנפילות 6.2.12

אם הזיכרון נמחק ולא סיימנו בצורה מסודרת (נפילת מערכת או השחתה של אזור בדיסק), משקמים מהדיסק את הכל. אם האינפורמציה לא שמורה טוב על הדיסק, נתקל בבעיה. מצד שני, לא ניתן לעדכן כל הזמן את הדיסק.
כי חשוב - שה־metadata יהיה עקבי, אחרת מערכת הקבצים עלולה להיות לא שמישה. ההנחה היא שהקובץ עצמו באחריות המשתמש, ה־metadata צריך להשמר ע"י מערכת ההפעלה, וכן המידע על הבלוקים הריקים והמלאים (היכן המידע שמור).
האחריות שלנו כמשתמשים - לדאוג שהמידע שלנו ישמר כל הזמן, כלומר לעשות save.
לעיתים נכפה $write-through$ - עדכון meta-data ייכתב מיד לדיסק. זה מאט את העבודה.
מידע חשוב לא נכתב באותו המקום, אלא נוצר עותק חדש, על מנת לאפשר התאוששות מגרסא קודמת אם הכתיבה השתבשה.
מידע עוד יותר חשוב - משכפלים, כדי שתמיד יהיה אפשר לחזור למידע בדיסק, כך שאם יש תקלה מקומית בדיסק אז עדיין יהיה אפשר לשחזר (לא מהפסקת חשמל).
בעת $boot$, מבצעים בדיקת עקביות והתאוששות - משווים את מצב הקבצים ואת עקביותם - כמו כאשר מחשב נייד של וינדואז לא נסגר טוב, ואז יש מסך שחור שרץ על הקבצים. ביוניקס יש תוכנה שאפשר להפעיל שתעשה זאת.
על מנת לפתור בעיות, השאילו מעולם ה־database פתרון נוסף:
שומרים בלוג מה רוצים לעשות, ורק לאחר סיום ביצוע השינוי מוחקים מהלוג. אז, אם יש נפילה במהלך ביצוע השינוי, אנו יודעים מה רוצים לעשות ואפשר לפעול לפי הלוג - לפעולה זו קוראים journaling.
חסרון - צריך שתי כתיבות במקום אחת. לכן עושים פעולה זו רק על מידע קריטי.
כיצד מבצעים?
לפני הכתיבה - כותבים בלוג רשומה מהו המידע שהיה ומה אמור להשמר. יש רשומות מיוחדות לתחילת וסיום טרטנזאציה.
לאחר הכתיבה בלוג, כותבים את הערך המעודכן למקומו בדיסק, כתיבה זו יכולה גם להיות $lazy$ - עצלה.

לזכרון\דיסק $flash$ ו־SSD מאפיינים שונים מדיסק מגנטי - יש קושי עם כתיבה מחדש באותו אזור:

- יש למחוק לפני כתיבה מחדש.
- המחיקה איטית, ויש לבצעה ביחידות גדולות.
- המחיקה פוגעת בהתקן, למשל לא ניתן למחוק יותר ממליון פעמים לאותו אזור.

לכן, הדרכים עליהן דיברנו עד כה לא טובות לשימוש בטכנולוגיות החדשות הנ"ל. מה עושים?
למשל LFS - מערכת קבצים מבוססת לוג - כל המערכת נשמרת כלוג: רצף לבלוקים הפנויים, רצף לבלוקים הכתובים. ה־$inodes$ מפוזרים בין רשומות בלוק ולא במקום קבוע בדיסק. גם הבלוקים של כל קובץ מפוזרים בלוג. בראש הרשימה הרשומות החדשות, בסוף הישנות, ואז אפשר לדעת מה ניתן למחוק. כל כתיבה יוצרת רשומה בלוג - יצירת בלוק או עדכון בלוק. כאשר צריך ליצור בלוק חדש, מקצה הרשימה לוקחים בלוק. הכתיבה היא של סגמנטים גדולים מהלוג, ואז מקבלים I/O רציף שמביאה לניצולת גבוהה של הדיסק (עם זאת, הקריאה פחות יעילה, אבל צוואר הבקבוק הוא כאמור בקריאה).
בקריאה: צריך למצוא את ה־inode במקום האחרון בלוג בו הוא נמצא. לכן מחזיקים מפה שממפה את מספר ה־inode למיקומו האחרון בדיסק. ב־LFS שומרים את המפה גם בדיסק וגם בזכרון, אולם במערכות $flash$ לא נשמרת מפורשות בדיסק כי בלאו הכי מייצרים את הטבלה בזמן אתחול.
יתרונות:

- כתיבה סדרתית ביחידות גדולות - פחות תזוזות בדיסק מכאני, פחות overhead ב־flash.

חסרונות:

- בניית מפת inodes איטית.
- כתיבת מפת inodes לדיסק הופכת כתיבה ללא סידרתית.
- יש מחיקה של מידע שלא צריך להמחק בגלל שהוא בסוף הלוג.
- קשה לדעת כמה מקום פנוי יש בהתקן.

6.2.13 מערכות קבצים מבוזרות

משותפות למספר מחשבים. לא היו לאחרונה שינויים גדולים, אולם יבואו בעתיד.
למה בכלל צריך? כביכול אפשר היה להשתמש ב־ftp לגישה לקובץ במחשב מרוחק, או ב־http - הבעיה: בשני המקרים הללו אין שקיפות, - כלומר, המשתמש מודע לכך שהמידע לא אצלו. מערכת קבצים מבוזרת נותנת אפשרות לקבלת מרחב שמות אחיד לאוספים גדולים של קבצים שלא נמצאים על הדיסק שלנו, ומאפשרת להעביר קבצים ממקום למקום.
מערכת אחת לנושא זה - AFS, אשר פותחה בשנות השמונים, משתמשת ב־session semantics - הקבצים שמורים במערכת אוניברסלית, כאשר משתמש רוצה לגשת הוא לוקח בעלות, והיא לא תראה מיידית למשתמשים האחרים המחזיקים את אותו הקובץ פתוח. כאשר המשתמש משחרר, השינויים נראים לכל הסשנים שהתחילו מאוחר יותר. יש כאן אוברהד גדול.
כיום משתמשים בעיקר במערכת NFS, שם הסמנטיקה לא מוגדרת, שכן העדכונים יכולים לקחת זמן לא מוגדר. כדי להבטיח קונסיסטנטיות יש לחשוב על מערכת מעליו. מערכת זו נפוצה מאוד ביוניקס\לינוקס.

NFS - Network File System NFS פותחה בשנות השמונים ויועדה למערכת קבצים של Sun - כל מכונה יכולה להיות כל לקוח וגם שרת. שרתים מפיצים ושומרים באמצעות export עצים שלמים. הלקוחות קושרים קבצים מהשרתים אל המערכות שלהם בפעולת *mount*. מדריך שהוא mounted בנקודה מסויימת בעץ של הלקוח הופך לחלק מעץ הלקוח (תת עץ). הלקוח מחליט איפה בהיררכיה שלו לתלות את ה־mount.
פרוטוקול מאונט:

- לקוח שולח בקשה.
- השרת שולח file handle המזהה את מערכת הקבצים - הדיסק, מספר ה־inode של המדריך, מידע לגבי הרשאות, וכו'.
- מפסידים כאן אכיפה של ההרשאות.
- ה־$handle$ ישמש לביצוע קריאה וכתיבה בתת העץ הזה.

automount: עושים מאונט כשיש דרישה - כלומר כשיש פניה לקובץ, מחפשים ברשת שרת שיענה לבקשה (יתכן ויש כמה).
יתרון - אפשר לשמור קבצי קריאה בלבד בעותקים רבים, זה מאפשר ניצול גדול יותר של המשאבים מבחינת מ"ה.
התוצאה - השגנו שקיפות:

- גישה לקובץ נראית למשתמש כמו גישה מקומית.
- המיקום הפיזי של הקובץ לא ידוע למשתמש.
- הזזת הקובץ מחייבת מאונט חדש.
- ה־$automount$ יודע למצוא אותו.

אפשר לחשוב על פרוטוקול שלא שומר מידע לגבי הלקוחות - stateless. הוא קל למימוש (אין צורך לשמור על קונסיסטנטיות מורכבת בין משתמשים שונים ברשת), התאוששות שקופה מנפילות שרת בזמן שהלקוח רץ. חסרונות - קשה לתמוך במנעולים, וכן קשה לשרת לבצע אופטימיזציות תלויות מצב (שכן אין מידע על המצב!). מימוש NFS - השרת הוא stateless (לא הלקוחות!): הוא לא תומך במנעולים.
איך ממשים גישות? משתמשים בשירות Remote Procedure Call (מפורט בהמשך) - הפעלת פונקציה במחשב מרוחק (שרת). בגלל תקלות, בקשות יכולות להתבצע פעמיים.
NFS מבצע *caching* מאסיבי על מנת לשפר ביצועים- נשמרים אצל הלקוח חלקים מהקובץ איתו הוא עובד. לא מובטחת סמנטיקה (בניגוד ל־AFS) - כלומר אם התבצע שינוי לא יודעים זאת. בקיצור נמרץ - טוב לשקיפות, רע לסמנטיקה ועקביות.

SMB- Server Message Block פרוטוקול המאפשר גישה לקבצים, מדפסות וכו' במחשבים אחרים. הוא תומך גם בתקשורת מאובטחת.

שיתוף דיסקים בדוגמאות הקודמות, לכל מחשב יש דיסק לוקאלי ומערכת קבצים משלו. כיום נפוצות מערכות בהן מחשבים משתפים דיסקים - למשל NAS - מחשב יעודי עם שטח אחסון גדול, המריץ תהליכי שרת של מערכות קבצים עבור לקוחות במחשבים שונים, ו־SAN, הנותן גישה ישירה לבלוקים בדיסקים משותפים ממספר מחשבים. שיטה זו מהירה יותר ויקרה יותר מ־NAS.

7 מערכות מבוזרות

7.1 מבוא למערכות מבוזרות

הגדרה 7.1 מערכת מבוזרת היא קבוצת מחשבים שלכל אחד זיכרון נפרד (לפחות) המקושרים ביניהם ברשת תקשורת ומשתפים ביניהם משאבים כלשהם.

המאפיינים של מערכת מבוזרת:

- מספר מחשבים, לכל אחד מעבד, זיכרון. יתכן דיסק, ממשק לסביבה, I/O
- רשת תקשורת
- "מצבים" משותפים - למשל תוכן קובץ, תור למדפסת משותפת...

יעדים למערכת מבוזרת להשיג את הטוב משני העולמות:

- מערכת הטרוגנית, בפריסה ריחבה, תומכת בגידול הדרגתי.
- נותנת אוסף שירותים אחיד - שמות, רישום משתמשים, זמן, קבצים...
- עם הבטחות גלובליות - שמות אחידים, זמינות מכל מקום, אמינות, ניהול פשוט, בטיחות.

עבור כל מכונה, משאב שנמצא באותה מכונה הוא לוקאלי - local, ומשאב שנמצא מכונה אחרת הוא רחוק - remote.

7.1.1 אבסטרקציות למערכות מבוזרות

- העברת הודעות בעזרת ערוצי תקשורת.
- remote login - מחשב אחד מגיע למשאבים של מחשב שני בצורה מרוחקת, למשל telnet,shh,rlogin, וכו'. כל הפרוטוקולים עובדים מעל תשתית של העברת הודעות. בדר"כ משתמש ב־TCP, מכיוון ויש דרישה לאמינות. במכונת היעד רצה תוכנית שרת המאזינה לפורט ידוע, לקוחות יוצרים ערוצים לפורט הזה.
- file transfer - ftp תומך ב־get/put וטיול בעץ של מערכת קבצים רחוקה.

כל האבסטרקציות מעלה לא שקופות - המשתמש יודע שהוא ניגש למערכת מרוחקת.

- מערכות קבצים מבוזרות - לקוח יכול לגשת לקבצים שיושבים במחשב אחר. ראינו כבר מערכות כאלה - חלק מהקבצים יושבים מקומית, חלקם במיקום מרוחק, ואפילו ניתן לקשר בין שתי המערכות.
- RPC - אריזה שנראית למשתמש כמו קריאה רגילה לפונקציה, במודל שרת־לקוח. ממומשת באמצעות stubs. מימוש בעזרת העברת הודעות UDP.
  - מגבלות:
    * העברת פרמטרים by reference בלתי אפשרית - מגבילים את סוגי הפרמטרים.
    * לפונקציה אסור לגשת למשתנים גלובליים - אסור שיהיו תוצאות לוואי.
    * ייתכן ייצוג שונה למספרים במחשבים שונים.
  - הלקוח לא מוצא את השרת? הקריאה מחזירה אקספשן מקומית אצל הלקוח.
  - הודעה אובדת? שולחים ACKs, מחכים timeouts, מבצעים שליחה מחדש, משתמשים ב־sequence numbers.
  - השרת נופל אחרי קבלת הבקשה?
    * ייתכן שביצע הפעולה לפני נפילה, ויתכן שלא.
    * אפשר לחכות ששרת חדש יעלה ולשלוח אליו את הבקשה.
    * צריך לבחור בין מודל של at-least-once: יתכן ביצוע כפול, או at-most-once: בוודאות אסור לנסות פעם שניה, ואז יכול שלא יבוצע בכלל. ניתן להוכיח שאי אפשר להגיע לאטומיות של "האם משהו בוצע או לא בוצע" במיקום מרוחק.

- הלקוח נופל אחר שליחת הבקשה? כלומר, התקשורת נפלה או הלקוח התנתק.
  * תופעת לוואי - אם היתה תקלה אצל הלקוח והוא מתעורר מחדש ושולח הודעה חדשה, צריך להיות מודעים לאי הרציפות של העבודה, וצריך למנוע בלבול בין בקשות חדשות לישנות.
    · דרך אחת לטיפול בבעיה: מגדירים באיזה דור הבקשה, ואז ניתן לבדוק אם הדור מתאים לדור הנוכחי או לא.
    · דרך אחרת לטיפול הבעיה: לכל בקשה יש "תאריך תפוגה".
  * יש לנקות אחרי שלקוח נפל, למשל לשחרר מנעולים שהוא מחזיק - garbage collection. תמיד יהיו שרכים, צריך לדעת להבדיל.

7.1.2 Kerberos - אימות במערכות מבוזרות

שירות אימות, המתבסס על שרת אימות בטוח ועל הצפנה. השרת יודע את כל הסיסמאות, אבל הם אף פעם לא מועברות ברשת. משתמשים בסיסמאות ליצירת מפתחות הצפנה. הסביבה של המערכת - משתמשים מחוברים לתחנות עבודה, המחובר לשרת אימות, ולשרת מיוחד זה מחוברים גם השרתים שיטפלו בבקשות של המשתמשים.
יש הפרדה בין שתי פעולות:

- אימות - כניסה ל"רשת".
- תקשורת - ניהול סשן בין שני צדדים.

תחנת העבודה של הלקוח בה המשתמש מנסה להתחבר, שולחת את שם המשתמש U לשרת.
שרת Kerberos מבצע את הפעולות הבאות:

- מחפש את הסיסמא של המשתמש - p, ומשתמש בפונקציה חד כיוונית על מנת ליצור מפתח הצפנה Kp ממנה.
- יוצר מפתח סשן חדש Ks.
- "אורז" את מפתח הסשן עם שם המשתמש: {U,Ks}.
- משתמש במפתח ההצפנה הסודי שלו Kk על מנת להצפין את הזוג.
- "אורז" את מפתח הסשן עם התעודה ticket שלא ניתן לפצחה (המפתח של שרת האימות לא ידוע לאף אחד), וכך יוצר את {Ks,{U,Ks}Kk}.
- לבסוף, הכל מוצפן בעזרת המפתח של המשתמש שיוצר על ידי הסיסמא של המשתמש, המביא ל-{Ks, {U,Ks}Kk}Kp.
- האחרון נשלח ללקוח.

התחנה של הלקוח מבצעת את הפעולות הבאות (המחיקות מוודאות שהסיסמא קיימת בזיכרון רק זמן קצר):

- מבקשת מהמשתמש את הסיסמא שלו p, ומחשבת בעזרתה את Kp, ומוחקת את הסיסמא.
- בעזרת Kp, התחנה של הלקוח מפענחת את ההודעה שהיא קיבלה מהשרת, וכך משיגה את Ks ו-{U,Ks}Kk.
- מוחקת את מפתח המשתמש Kp.

כעת, התחנה של הלקוח יכולה לשלוח בקשות אימות לשרת Kerberos. כל בקשה תורכב משני חלקים:
הבקשה עצמה R, מוצפנת בעזרת Ks, והתעודה שלא ניתן לפצחה. השרת מפענת את התעודה בעזרת המפתח הסודי שלו Kk, מוצא את U ואת Ks.
שרת Kerberos לא מספק שום שירותים אמיתיים - כל מה שהוא עושה הוא לספק מפתחות עבור שרתים אחרים. הוא ישלח את המפתח המקוצה Kf ללקוח המוצפן בעזרת Ks, וכן ישלח אותו לשרת הקבצים בעזרת Kb. הלקוח אז יוכל להשתמש ב-Kf כדי "לשכנע" את שרת הקבצים בזהותו, ולבצע פעולות על הקבצים.

7.1.3 על אמינות וזמינות במערכות מבוזרות

במערכות מבוזרות ללא יתירות האמינות והזמינות נמוכות יותר מבמערכות לא מבוזרות - מספיק שאחד הרכיבים יפול על מנת שהשרות לא יהיה זמין. על כן, מערכות אמינות משתמשות ביתירות:

- יתירות בשמירת מידע על דיסקים RAID
- יתירות ברכיבים הפיזיים
- שמירת מספר עותקים של מידע read-only
- לעיתים יש גם שכפול בכמה שרתים של המידע שנכתב

7.2 וירטואליזציה

מכונה וירטואלית - תוכנה רצה מעל חומרה, ויוצרת אשליה של חומרה אחרת. הממשק זהה לזה של חומרה הפיזית. יכולה "לחלק" את החומרה למספר מכונות.

סיבות לשימוש

- ניצול משאבים:
  - קונסולידציה - הרצת שרתים שונים על חומרה אחת.
  - חלוקת משאבים ב־cloud - ניתן לשלם על חלקי מכונה, למשל שני מעבדים לשלוש שעות ו10 ג'יגה תעבורה ברשת.
- ניהול שרותי מחשב:
  - שכפול סביבות מלאות (התקנה מלאה כולל מ"ה) - הקמת סביבות חדשות זהות לקודמות.
- פיתוח רכיבי מ"ה:
  - בדיקה על מגוון חומרות.
  - דיבג מבלי להפיל את המכונה - פשוט מריצים מחדש את האפליקציה, אין צורך בריבוט.

7.2.1 סוגי מכונות

$Hypervisor/VMM$ היא תוכנה המממשת מכונה וירטואלית. סוגים שונים:

- $bare-metal$ - רץ ישירות על החומרה, למשל VMware, ESX, XEN.
  - יתרונות:
    - שליטה מלאה בחומרה עצמה.
    - אין תחרות עם מ"ה.
- hosted - רץ מעל מ"ה לצד תהליכים רגילים, למשל VMware Workstation.
  - יתרונות:
    - $hypervisor$ רזה יותר, לא צריך לממש מחדש מנגנונים של מ"ה (זימון, ניהול זיכרון).
    - ניתן להריץ תהליכים רגילים לצד VM.
    - ניהול פשוט של VM עם כלי לינוקס סטנדרטיים.
- שילוב - בעיקר הוסטד, אבל יש גם קוד בקרנל של מ"ה המארחת (לשיפור ביצועים).

מ"ה היא לתהליכים מה ש־hypervisor הוא למערכות הפעלה - ה־hypervisor צריך לספק אבטרקציה של חומרה ולנהל משאבים.

7.2.2 תרגום כתובות זיכרון

איך מתורגמת הכתובת הוירטואלית, לכתובת פיזית של ה־guest VM, לכתובת הפיזית של ה־host?
ה־$hypervisor$ מגדיר shadow page table.

- תרגום כתובת וירטואלית לכתובת הפיזית של מכונת ה־host - מדלג על רמת הביניים.
- תרגום מהיר, אבל באיזה מחיר? עדיין עובדים על זה.
- כל עדכון של טבלת הדפים באורח חייב להשליך על עדכונים ב־hypervisor.
- כאשר מחליפים את ה־VM הרץ - דרוש TLB flush.

פתרון: היום יש חומרה התומכת ב־nested page tables. שדה ASID ב־TLB מציין לאיזה VM קשורה כל שורה.

7.2.3 דימוי CPU

- שיטה 1 ־ אמולציה של כל פקודה.
  - נדמה את פעולת המעבד בתוכנה. למשל, נחזיק מערך של כל הרגיסטרים במעבד, בכל כתיבה לרגיסטר, נכתוב למערך.
  - מקבלים האטה אדירה, אבל הרבה ארגונים מתחילים לעבור לסביבה הזו בשל ה"נקיון" שלה ־ אין צורך לבדוק מה תהליכים השאירו מאחור.
- שיטה 2 ־ Hardware Accelerated Virtualization
  - רב הפעולות של ה־VM רצות ישירות על המעבד.
  - יש חומרה חדשה המאפשרת ל־$hypervisor$ להתערב כל פקודה רגישה.
  - יש בחומרה רמת הרשאות מיוחדת ל־hypervisor, גבוהה יותר מזו של הקרנל של המארח, בכל פקודה שדורשת התערבות hypervisor־ trap.
  - מגביר את היעילות של הפקודות האלו.
- שיטה 3 ־ תרגום קוד בינארי דינמי
  - לפני הרצת קטע קוד (בינארי) תרגום לקוד "בטוח".
    - רב הפקודות מתורגמות לעצמן ־ רצות ישירות על המעבד.
    - פקודה שדורשת התערבות hypervisor מתורגמת ל־hypercall ־ המקבילה של system call ב־hypervisor (מבצעת trap).
  - $VMware$ עובד בצורה זו.
- שיטה 4 ־ $Paravirtualization$
  - מריצים בתור $guest$ רק מ"ה המותאמות לריצה מעל hypervisor.
    - לא קוד מקורי של מערכת הפעלה.
    - למשל, במקום לחסום פסיקות, מבצעות hypercall להודיע ל־hypervisor שלא רוצות לקבל פסיקות.
  - לא מאפשר ריצות של מ"ה מסויימות.
  - Xen עובד כך.

8 הקדמה ל־Security

8.1 בעיות אפשריות

Eavesdropping and packet sniffing תיאור: השגת מידע מבלי לשנותו.
אמצעים: packet sniffers, ראוטרים, gateways, תפיסה ופילטור פאקטים.
איומים: אפשר להשתמש ב־sniffing על מנת לתפוס מידע מגוון הנשלח ברשת ־ לוגין וסיסמא, מספרי כרטיס אשראי...

Snooping תיאור: השגת מידע מבלי להשאיר עקבות.
אמצעים: מעבר על קבצים בדיסק או על הזיכרון הראשי:

- על ידי שימוש בהרשאות לגיטימיות (פנימי).
- פריצה למערכת (חיצוני).
- גניבת מחשבים ניידים.
- ניטור הקשות מקלדת.
- observing timing information (convert channels).

איומים: השגת מידע רגיש (קבצים עם מספרי כרטיס אשראי, למשל), גילוי סיסמאות, מפתחות סודיים, וכו'.

Tamparing תיאור: שינוי או השמדת מידע שמור.
אמצעים: שימוש בלתי הולם בהרשאות (פנימי) או פריצה חיצונית למערכת.
איומים: שינוי רשומות (למשל ציונים, רשומות חוב), שתילת סוסים טרויאנים, ועוד.

Spoofing תיאור: התחזות למשתמשים או מחשבים אחרים על מנת להשיג הרשאות.
אמצעים:

- גניבת חשבונות, ניחוש סיסמאות, social engineering.
- IP spoofing: זיוף אימייל, IP שקרי מכתובת, חטיפה.
- IP connections.

איומים: הודעות מזוייפות, סירוב מתן שירות.

Jamming תיאור: ניטרול מערכת או שירות.
אמצעים: שליחת בקשות (לגיטימיות) מרובות למארח עד לכילוי המשאבים.
איומים: חיסול כל המשאבים על המכונות המותקפות, ניצול באג כדי לכבות hosts.

Code Injection תיאור: "הזרקת" קוד זדוני לביצוע על host בעל הרשאות רבות והדבקת hosts אחרים.
אמצעים: וירוס, תולעת.
איומים: כולם...

8.2 שיטות

ניצול פגמים ניצול נקודות פגיעות בתוכנה על מנת לחדור למערכות.

- buffer overflow.
- פגמים באבטחת mobile code.

הידע מתפשט יותר מהר מה"תרופה".

פיצוח סיסמא ומפתח

- ניחוש ססמאות חלשות.
- התקפת מילון: חיפוש סיסטמטי.
- חיפוש מקיף:
  - כלים לניתוח קריפטוגרפיה מתפתחות כל הזמן.
  - האינטרנט מספק משאב מקביל.
- ניתוח קריפטוגרפי, יוצרי סיסמאות ומפתחות גרועים, ניתוח תזמון.
- פיצוח smart-card בעזרת fault injection.

Social engineering

- בניית מערכת מזוייפת: מסך לוגין, מספרי טלפון...
- בניית שירות מזוף: גניבת מספרי אשראי ו־PIN, גניבת סיסמאות.
- Agent-in-the-Middle attacks.

8.3 Buffer overflow

קורה לעיתים קרובות ב־stack או ב־heap. משמעות: overwriting על control-data או מידע רגיש.
כיצד פותרים?

- ברמת מערכת ההפעלה:
  - Exec shield.
  - Address space layout randomization.
  - ועוד.
- ברמת התוכניתן:
  - fget (no gets).
  - strncpy (not strncpy).
  - ועוד...

9 I/O ותקשורת בין תהליכים

9.1 מערכות קלט\פלט - I/O

התקני I/O מחוברים דרך ממשקים ו־buses. יש להם יש בדרך כלל חלק אלקטרוני - בקר\controller, או מתאם\adapter, וחלק מכני. הבקר מקבל מידע מההתקן ומתקשר עם המעבד דרך הממשק. בקר אחד יכול לנהל מספר התקנים פיזיים זהים.
ההתקן מכיל רגיסטרים מיוחדים לתקשורת עם המעבד. ישנן שתי שיטות גישה אליו:

- ההתקן מחובר לפורט - port - כתובת ההתקן ב־bus. הגישה - בעזרת פקודות אסמבלי מיוחדות.
- התקן ממופה לזיכרון - memory-mapped I/O - גישה מה־CPU להתקן דרך כתובות במרחב הכתובות (גם וירטואליות וגם פיזיות ממופות להתקן). בשיטה כזו התכנות קל - לא צריך פעולות אסמבלר מיוחדות, ומשתמשים בהגנה של טבלת הדפים. כמו כן, חוסך העתקת הרגיסטרים של בקר אל\מהזיכרון, כי ניתן לבצע פעולות ישירות אליהם.

9.1.1 סוגי התקנים

- התקני בלוקים - העברה ביחידות של בלוקים. לכל מידע יש כתובת, ויש גישה אקראית - למשל דיסק, CD.
- התקני תווים Character - העברת מידע ביחידות של בתים. אין לכל מידע כתובת, ואין גישה אקראית - למשל עכבר, מקלדת.
- התקני רשת - אין לכל מידע כתובת, ואין גישה אקראית, אבל מעבירים מידע לרוב ביחידות גדולות - חבילות תקשורת.

9.1.2 ניהול ההתקנים

תפקידי מערכת ההפעלה - ניהול משאבים, וסיפוק אבסטרקציות כבור גישה.
הבעיה: יש ניגוד בין כלליות, לבין יעילות הדורשת התאמה לכל התקן. הגישה הכללית: עבודה ברמות: ממשקים אחידים להתקנים מסוגים דומים.
מגדירים מטה־פקודות לפעולות עם אמצעי I/O, את רובן אנחנו כבר מכירים:

- get/put להתקני character.
- read/write/seek להתקני בלוק.
- socket להתקני רשת - ממשק שמסתיר את מאפייני הרשת.

תקשורת עם התקנים נעשית ע"י device drivers - לכל סוג התקן יש device driver משלו, והם שונים מהותית זו מזו. ההתקן ה־device רץ בקרנל מוד, ומשתמש בממשקים שונים של מערכת הפעלה, כלומר הוא יודע באיזו מ"ה הוא רץ, ומשתמש בממשקים הללו כדי להתקשר. למשל משתמש בממשקים בשביל הקצאת זיכרון דינמי, תפיסת פורטים, הפעלת interrupt handler לטפל בפסיקה.
devices צריכים לדאוג ל:

- הפעלת ההתקן.

- ניהול כח - ניתן להעביר התקן ל-stand by.
- הקצאה\שחרור משאבים אוטומטית.

כיצד מעבירים ומקבלים מההתקנים מידע? ישנן שתי גישות עיקריות, לפעמים עוברים בין שתי הגישות בהתאם לפעולות שמבצעים:

- דגימה - polling (לפעולות קצרות מאוד, או לפעולה שיש צורך בניטור כל הזמן):
  - ניטור כל הזמן, למשל במערכות רפואיות.
  - ה-CPU "עסוק" בזמן המתנה, אין context switch - לכן חשוב להפסיק להמתין כדי שמערכת ההפעלה לא תתקע. בשביל זה עושים לולאה.
- "שגר ושלח" - קלט-פלט מונחה פסיקות:
  - ה-CPU שולח בקשה להתקן, ועובר לעשות משהו אחר בזמן שההתקן עובד.
  - כאשר המידע זמין, ההתקן מייצר פסיקה.
  - מיושם ע"י שתי פונקציות:
    * פונקציה המבקשת את המידע מההתקן.
    * פונקציה שמטפלת בפסיקה - בדר"כ נעשה ע"י interrupt אבל לא תמיד (לפעמים יהיה סימון בזיכרון, ומתעורר תהליך שיבדוק את הסימון מידי פעם - הרווח: אפשר להחליט שהתהליך יתעורר בסוף תהליך אחר, או בשלב פחות קריטי).
  - יתרון: אפשר להריץ תהליכים אחרים תוך כדי.

איך מעבירים מידע בין התקן ל-CPU? שתי אפשרויות:

- MMIO or Programmed I/O - ה-CPU מעביר את המידע בלולאה.
  - פקודות in/out (או mov במקרה של MMIO).
  - בעיקר להתקנים המצריכים תגובה מהירה (character devices).
- Direct Memory Access (DMA) - "שגר ושלח" - מגדירים איזור מיועד להתקן, ההתקן מעביר את המידע ישירות לזיכרון או ממנו, ומתעורר כאשר קבלת המידע הסתיימה.
  - מתאים להתקני בלוק המעבירים מידע בקצב גבוה (למשל דיסק, כרטיסי קשת, כרטיס גרפי, כרטיס קול).
  - צורת הניטור הטבעית - interrupt בסיום הפעולה.
  - סדר הפעולות של DMA:
    * ה-CPU יוזם DMA ע"י כתיבת בקשה במקום קבוע בזיכרון ההתקן. הבקשה מציינת כמה מידע להבין, מהיכן ולאן.
    * כאשר המידע זמין, בקר ההתקן תופס את ה-bus ומתחיל העברת מידע לזיכרון. החומרה דואגת להעברה. כדי לייעל את העבודה, אפשר לבצע *prepatching*.
    * כאשר העברת המידע נשלמה, הבקר מייצר פסיקה.

9.2 Disk Caching

כאשר משתמש פונה לבלוק, הוא מצפה למצוא אותו בזיכרון. הבקשה עוברת דרך כל הרמות:

- בהתאם למיקום הראש קורא\כותב יוחלט מתי יקרא כל בלוק מה-$device$.
- רמת העברת הבלוקים עצמה - מערכת הקבצים מקבלת בלוקים ומעבירה לזיכרון. ליעול, יש בזיכרון איזור cache להעלאת דפים מהדיסק - disk cache. ממנו הוא עובר לדפים הספציפיים שמשתמש רוצה. מיד נפרט מדוע העבודה מתבצעת כך.
- *generic block layer* - מסתירה את המבנה הייחודי של כל דיסק.
- I/O scheduling layer - זימון גישות לדיסק למזעור seek time.
- דיסקים מסוגים שונים - זמן גישה איטי, קצב העברה מהיר.

בגלל ההבדל העצום בין מהירויות גישה לדיסק לבין גישה לזיכרון (5־6 סדרי גודל), caching של דיסק קריטי לביצועים.
disk cache ־ הדרך לעבודה יעילה: איזור זיכרון שאליו שופכים מ־block devices וממנו מוציאים מידע אל הדיסק. בבקר עצמו של הדיסק יש גם *caching* ־ כלומר דיסקים "שופכים" את הדפים שיש לכתוב אל ה־caching של הבקר. buffer cache מכיל מידע הכולל חלקי קבצים, מידע גולמי מההתקנים, דפים מזיכרון וירטואלי שהורדו לדיסק. כל ה־I/O עובר דרכו, מידי פעם דפים נזרקים או נכתבים לבלוק.
כאשר יש בקשות קריאה, הבקר בודק אם המידע ב־cache, אם לא, קוראים בלוק שלם מהדיסק, ומעתיקים את מספר הבתים הרצוי למקום הרצוי בזיכרון.
caching יעיל כי:

- לוקאליות ־ תהליך שקורא מבלוק לעיתים קרובות קורא בהמשך קטעים נוספים מהבלוק.
- קבצים משותפים נפוצים נשארים ב־cache, למשל קוד המורץ ע"י הרבה תהליכים.
- משתמש לעיתים ניגש לקובץ מספר פעמים ־ מעתיק קובץ, עורך אותו, מקמפל, שומר עותק כיבוי, עורך שוב...
- הרבה מידע שנכתב לקבצים הוא "חולף" ולא נכתב לדיסק בסופו של דבר.
- ניתן לבצע pre-fetching.

ע"י ניתוח שימוש מגיעים לגודל buffer cache אופטימלי.
היום ניתן להחזיק מיליוני דפים ב־cache, מחפשים לפי מקום בדיסק או מרחב כתובות אליו שייך הדף. יש לנהל את הזיכרון כדי לדעת מה נמצא שם.
החלפה של דפים ב־*cache* נעשית בצורה מודעת. אפשר למשל להשתמש ב־LRU אמיתי.
מבחינים בין שתי גישות לסנכרון מידע מול הדיסק:

- write-through: כל מידע שמעודכן נכתב מיד לדיסק.
  - איטי, כי נעשה תוך כדי כתיבה אל הדיסק.
  - היה מאוד פופלרי בעבר כשיהיה fluppy disk, כי לא היה ידוע מתי המשתמש יוציא את הדיסק.
  - דורש הרבה יותר disk I/O, ולכן פחות יעיל.
- write-back: עדכון בדיסק נעשה במועד מאוחר יותר.
  - השיטה הדומיננטית היום, שכן הסיכוי לטעות יחסית קטן, והשיטה מהירה הרבה יותר.
  - כתיבה משנה את העותק ב־cache. על כל בלוק מסמנים אם הוא *dirty* או לא, לציון כי המידע השתנה אך לא נכתב עדיין.
  - מידי פעם תהליך עובר וכותב דפים "מלוכלכים" לדיסק, ומכבה את הביט.
  - אפשר להכריח את המערכת לכתוב מידע מסויים לדיסק:
    * sync ־ יוזם כתיבה לדיסק וחוזר לפני שהכתיבה מתבצעת (כמו כשרוצים להוציא USB ־ מודיעים זאת למערכת, ולאחר זמן מסויים מופיעה הודעה כאשר ניתן להוציא אותו).
    * fsync ־ כותב לדיסק לפני החזרה.

למידע קריטי משתמשים בשיטה הראשונה, לכל השאר ־ בשיטה השניה.
חסרונות של עבודה עם caching:

- העתקה נוספת.
- write-through פוגע משמעותית בביצועים, write-back גורם לאובדן מידע בנפילות.

9.3 תקשורת בין תהליכים באותו המחשב

תהליכים יכולים לתקשר בעזרת העברת הודעות ע"י system calls.
עבור תהליכים באותו המחשב, נכיר שתי סוגי אבסטרקציות לתקשורת:

9.3.1 תיבות דואר

האבסטרקציה: לכל תהליך יש תיבת דואר אליה אפשר לשדר הודעות שרוצים להעביר אליו.
ב־$Mach$ כל התקשורת בין התהליכים מתבצעת בעזרת אבסטרקציה זו, כולל system calls ־ כאשר תהליך נוצר, יש לו שתי תיבות דואר מיוחדות ־ האחרת קרנל, לשליחת בקשות למערכת ההפעלה, השניה notify לשם הקרנל שולח הודעות על אירועים שהתרחשו.

9.3.2 ערוצי תקשורת שונים

מ"ה מספקת את האבטרקציה דרכה ניגשים לערוץ התקשורת בין התהליכים - למשל pipe ו-socket (עוד בהמשך). האבסטרקציה מיוצגת ע"י file descriptor, עליו ניתן לבצע פעולות שליחה וקבלה של הודעות.
אפשר להעביר בשתי דרכים:

- אסינכרונית: למקבל אין שליטה על הזמן בו המידע מועבר. קריאות חוזרות (לא חוסמות) ואומרת שנכשלו במקום להמתין. בשלב מסויים הצד השני יקבל את המידע.
- סינכרונית: המקבל מבקש את המידע באופן מפורש.
  - קריאות חסומות: המקבל מחכה עד שכל המידע נשלח.
  - קריאות לא חסומות: המקבל לא מחכה, מקבל מידע רק אם הוא כבר נשלח. קריאות חוזרות ואומרות אם הן נכשלו במקום.

  בדר"כ המקבל יכול להחליט בין קריאות חסומות או לא חסומות. אם הקיבולת של הערוץ מוגבלת, כשאין מקום בערוץ, send ממתין עד שיהיה מקום ואז מתבצע.

בתרגול ראינו דרכים ספציפיות לשימוש בערוץ תקשורת בין תהליכים על אותו המחשב ב-Unix. בחירה באיזה מכניזם להשתמש תלוי בצרכים.

- סיגנלים: תהליך יכול לשלוח סיגנל לתהליך אחר.
  - הסיגנל נשלח ישירות לתהליך.
  - על השולח לדעת את ה-id של התהליך המקבל.
  - א-סינכרוני - הסיגנל מתקבל ללא בקשה מצד התהליך המקבל.
  - ההודעות מוגדרות מראש (ללא מידע).
  - אין persistence (ערוץ התקשורת לא נשאר פתוח).
- pipes: stream חד צדדי בין תהליכים הקשורים זה לזה.
  - סינכרוני.
  - persistence ברמת התהליך.
  - כך ה-shell מיישם input/output redirection.
  - על מנת שתהליכים יוכלו לחלוק pipe, עליהם להיות קשורים זה לזה - כלומר, יוצרו בעזרת fork או exec מאותו תהליך אב.
  - יש לו בדר"כ קיבולת מוגבלת.
  - באופן דיפולטיבי, פעולות read(), write() חוסמות.
- FIFO: נקרא גם named pipe.
  - השם מרשה גם תקשורת בין תהליכים שאינם קשורים לתקשר דרכו.
  - למעשה הוא קובץ מיוחד, על כן ה-persistence שלו היא ברמת מערכת הקבצים.
  - מספר תהליכים יכולים לבצע open(), write(), read().
  - פתיחה לכתיבה חוסמת עד שמישהו פותח לקריאה, וכנ"ל בכיוון השני.
  - אם הקורא סוגר אותו, הכותב מקבל SIGPIPE כאשר הוא מנסה לכתוב.
- תורי הודעות Message queues: דומה לקודם - רשימה של הודעות בעלת key מפתח מיוחד המצורף לה.
  - כל תהליך יכול לשלוח הודעה לתור או לקבל הודעה מהתור, כל עוד הם יודעים מהו המפתח המזהה.
  - persistence ברמת הקרנל (בניגוד ל-FIFO).
  - אין צורך לפתוח או לסגור אותו.
  - להודעות יש גדלים - המקבל מקבל את כל ההודעה שנשלחה ע"י השולח.
  - להודעות יש סוגים - המקבל יכול לבקש סוג הודעה מסויים, ובמקרה זה ההודעות לא יתקבלו בסדר בהן נשלחו.

- msgget יכול לבחור ליצור את התור אם לא קיים, מחזיר את ה־msqid.
- המפתח נבחר ע"י סוג של פונקציית האש על קובץ מוסכם כלשהוא - ftok. כך תהליכים שונים יכולים להפעיל את הפונקציה ולקבל את אותו המפתח לשימוש. כדי להפעיל הפונקציה, לתהליך צריכה להיות הרשאת stat על הקובץ.

- סגמנטי זיכרון משותפים:
  - תהליכים יכולים לבקש ממערכת ההפעלה סגמנט של זיכרון משותף לשימוש להעברת הודעות.
  - זיהוי בעזרת מפתחות, כמו הקודם, אפשר להשיג באותו האופן.
  - אסינכרוני - הכותב יכול לשנות את הזיכרון המשותף ללא שליטה של הקורא. בשביל סינכרוניזציה, יש להשתמש בסמפורים.
  - persistence ברמת הקרנל.
  - התקשורת מאוד מהירה, אין צורך ב־system call לקריאה או כתיבה.
  - מחיקה של הסגמנט מתרחשת רק אחרי שכל התהליכים המחוברים עליו התנתקו.

- יש דרכים נוספות, כמו memory mapped files, וסוקטים, בהם גם משתמשים עבור תקשורת ברשת.

9.4 תקשורת בין תהליכים במחשבים שונים

9.4.1 שני פרוטוקולי תקשורת

אפשר להעביר הודעות גם בין תהליכים במכונות שונות, בעזרת פרוטוקולי תקשורת מוסכמים.
על הפרוטוקולים הנפוצים, בקצרה (עוד בהמשך):

TCP	UDP
stream - רצף בתים	datagrams - רצף חבילות
FIFO	לא מסודר
ניתן לסמוך עליו	אין הבטחה שהמידע הגיע
ה־ACK כבר בפנים	best effort - אם רוצים אמינות יש לדאוג לה לבד

9.4.2 המודל

כיוון וגודל הבאפר מוגבל, ומסיבות אחרות, לא ניתן לשלוח הודעות שאורכן ארוך באופן שרירותי. אבל המשתמשים לא יודעים זאת, ואולי רוצים לשלוח ספר שלם בהודעה יחידה. על כן, יש לפרק הודעה ארוכה ל־packets קטנים בצד השולח, ולחבר אותם בחזרה להודעה אחת ארוכה בצד המקבל. תהליך זה נקרא packetization.

נוח לחלק את הביצוע של התקשורת למספר רמות. כל רמה נבנית על הרמות מתחתיה ונותנת שירות נוסף לכל הרמות מעליה, ולבסוף למשתמשים. הפאקט שמועבר מחולק לשני חלקים: המידע, והדר - header. המידע מועבר מטה דרך הרמות השונות, ומשיג הדרים נוספים בדרכו בצד השולח. הוא אז מועבר דרך הרשת, ולבסוף עולה מעלה בצד המקבל, וההדרים "מקולפים".

הסט של התוכנות שההודעה עוברת דרכם נקראות protocol stacks. לוגית, כל שכבה מדברת ישירות עם השכבה המתאימה במכונה השניה בעזרת פרוטוקול מסויים. כל פרוטוקול מציין את הפורמט והמשמעות של ההודעות שיכולות לעבור. לרב ההדר הוא סציפי לפרוטוקול מסויים, ושאר ההודעה לא מתורגמת.

הפרוטוקול גם מניח הנחות מסויימות על התכונות של תת־מערכת התקשורת שהוא משתמש בה. למשל, אפליקציית האימייל מניחה כי כל ההודעה מועברת ללא טעויות. במציאות, האבסטרציה הזו של תת־מערכת התקשורת נוצרת על ידי השכבות הנמוכות יותר בסטאק זה.

על מנת שיהיה אפשר לתקשר, מחשבים צריכים להחליט על המבנה הפונקציונלי של ה־protocol stacks, ועל הפורמטים של ההדרים בה משתמשת כל רמה. זה מביא ליצירת מערכות פתוחות, שיכולות לקבל מידע חיצוני ולהטמיע אותן בצורה נכונה.

כך נוצר מודל סטנדרטי על ידי ארגון התקינה הבינלאומי - ISO-OSI. הרמות בו הן:

- Application - יישום: ממשק לתקשורת עם המשתמש.
- Presentation - הצגה: טיפול בייצוג המידע, כלומר קידוד ודחיסה.
- Session - שיחה: שליטה על הדיאלוג (לא בשימוש).
- Transport - תעבורה: פקטיזציה ואמינות. אם node נכשל בדרך, הבדיקות ימצאו זאת וישלחו מחדש את הפאקט. רמות גבוהות יותר יכולות להניח כי הקשר אמין.

- Network - רשת: ניתוב - העברת מידע ברשת מקצה לקצה. זוהי השכבה העליונה ביותר בה משתמשים ב-routing nodes. רמות גבוהות יותר לא צריכות לדעת כלום על הרשת.
- Data Link - קו: העברת נתונים מנקודה לנקודה באופן נכון למרות הפרעות. רמות גבוהות יותר יכולות להניח כי ניתן לסמוך על אופן העברת המידע.
- Physical - פיזית: העברת אותות, הגדרת מתחים, הגדרת חיבורים, וכו'.

9.4.3 פרוטוקולים בסוויטת TCP/IP

פרוטוקולי IP הם הסטנדרט דה־פאקטו באינטרנט פרוטוקול האינטרנט - IP, מתעסק בניתוב מידע דרך מספר רשתות, ועל כן לוגית מאחת את כל הרשתות הללו לרשת אחד גדולה. זו נקראת שכבת ה"רשת", ו־IP הוא פרוטוקול הרשת. עם זאת, רשתות יכולות להשתמש בשירות ראוטינג משלהם, אז אפשר לומר כי IP מתאים לשכבה הגבוהה ביותר של הרשת. הרמות מתחת לזו - רמת הקו והרמה הפיזית, אינן חלק מסוויטת ה־TCP/IP - במקום זאת, נעשה שימוש במה שזמין בכל מערכת.
TCP ו־UDP מספקים end-to-end services (בהתבסס על IP), והופכים אותם לפרוטוקולי תעבורה. בנוסף, הם אחראים להעברת המידע לאפליקציה הנכונה, זאת בעזרת קישור של כל אפליקציה למספר פורט.
השכבה העליונה היא שכבת האפליקציה. מספר אפליקציות בסיסיות הן חלק אינטגרלי מסוויטה זו, ועל כן הן זמינות ברחבי העולם.

השדות ב־IP header

- גרסת הפרוטוקול.
- אורך ההדר.
- אינדיקציה לאיכות השירות המבוקשת.
- אורך הפאקט.
- שלושה שדות המטפלים בפרגמנטציה.
- "זמן חיים" - תוך כמה "קפיצות" יש להתעלם מהפאקט הזה.
- מזהה של הפרוטוקול ברמה הגבוהה יותר - TCP או UDP.
- checmsum של ההדר, לוודא שהמידע בו לא ניזוק.
- כתובת ה־IP של השולח.
- כתובת ה־IP של היעד.

פרוטוקול UDP פרוטוקול זה מספק גישה ישירה ל־datagrams - היחידות ה"אטומיות" בהן המידע מועבר ע"י IP. עם זאת, יש רשתות בהן היחידה שמועברת קטנה יותר. על כן, על IP לפרק ולהרכיב מחדש את ה־datagrams לאורך הדרך.
כאמור, פרוטוקול זה מספק גישה ישירה ל־datagrams, וכך מרשה למשתמשים לשלוח datagrams עד לגודל קבוע מסויים, ללא תקורה נוספת עבור אמינות. בשיטה זו, הן נשלחות בתקווה שידיעו ליעדן. עם זאת, checksum כלול בהן, כך ש־datagrams שניזוקו בדרך יאותרו. ל־UDP יש יכולות נוספות שאין ל־TCP, כדון יכולת broadcast.

פרוטוקול TCP פרוטוקול זה הוא פרוטוקול אמין. הוא יוצר קישור מתמיד בין האפליקציות, כך שהאפליקציות יכולות לקרוא ולשלוח מידע דרך קישור זה, בצורה לא מובנית - רצף בתים. הוא מפרק את ה־datagrams, ומוודא כי הן הגיע לצד השני באופן תקין ומופיעות בסדר הנכון. על מנת לעשות זאת, הוא שומר כל datagram בבאפר עד שמתקבלת ההודעה כי היא הגיעה ליעדה - acknowledge - ACK.
השדות ב־header של ה־TCP הם:

- מספר הפורט של המקור.
- מספר הפורט של היעד.
- מספר הרצף של ה־byte הראשון של מידע שנשלח בפאקט הזה.
- acknoledgments - מופיע כמספר הרצף של ה־byte הבא שהשולח מצפה לקבל מהמקבל.
- גודל ה־header, כדי לדעת היכן המידע מתחיל. לאחר מכן4 ביטים שאין להם שימוש מיועד.

- שמונה פלאגים באורך ביט, עבור שליטה (למשל יצירת קישור חדש).
- המקום הפנוי של השולח לקבלת מידע מהמקבל.
- checksum על כל הפאקט.
- אינדיקציה לדחיפות המידע.

9.4.4 איתור ותיקון שגיאות

ביט בדיקת זוגיות ביט זה הוא פשוט הסכום של ביטים מסויימים - נקרא כך משום שבייצוג בינארי הוא בעצם מייצג את הזוגיות של הביטים האלו. אם נוסיף רק ביט בדיקת זוגיות לסטרינג של ביטים, אם נפלה טעות אחת, אפשר לזהות. אם שתיים, אז הזוגיות "תתאפס" ולא נמצא שנפלה טעות.

קוד האמינג $11,7$ לכל 7 ביטים של מידע מוסיפים 4 ביטים של בדיקת זוגיות. כל ביט בדיקת זוגיות נמצא במקום של חזקת 2, לכן בייצוג המיקום שלו בבינארית יש ספרה אחת שאינה אפס. כל ביט בדיקת זוגיות בודק את הזוגיות של הביטים שבמספרם בבינארית יש 1 במקום של ביט הזוגיות. בעזרת קוד האמינג אפשר לתקן טעות אחת. אבל שוב אנו נתקלים בבעיה - למשל אם שני ביטים שגויים, הביטים של בדיקת הזוגיות לא יעבדו כמו שצריך.

קוד מתוחכם יותר - CRC נביט במקרה בו מקודדים את ההודעה כמספר. נוסיף להודעה עוד מספר ביטים, כך שאם נחלק את המספר המתקבל במספר קבוע מראש, לא תתקבל שארית. זו הפעולה שהמקבל עושה, אם אין שארית, ההנחה היא שההודעה תקנית, אחרת נפלו בה שגיאות.

מה קורה אם הפאקט כלל לא הגיע? כאשר פאקט מגיע ליעד, המקבל שולח ack שהמידע הגיע, ואם הגיע משובש שולח nack, וכך השולח יודע לשלוח בשנית. אם הפאקט כלל לא הגיע, השולח מחכה להודעה שלא תגיע, ולא ידע שהחבילה לא הגיעה. כדי לפתור את הבעיה, קובעים חלון זמן לקבלת הודעה מהמקבל. אם לא התקבלה עד פקיעת השעון, השולח ישלח שוב את הפאקט. אבל, מה אם הפאקט פשוט התעכבה בדרך? או אולי אישור הקבלה התעכב? בשביל זה לכל פאקט יש מספר סידורי, לפיו המקבל יכול לדעת אם כבר קיבל את הפאקט או לא, וכך יכול להתעלם ממנו אם הוא מגיע מספר פעמים.

9.4.5 Buffering and flow control

....

9.4.6 שליטה בעומס ב־TCP

שליטה שכזו בעומס היא מקרה מיוחד של ניהול משאבים ע"י מערכת הפעלה. יש לה מאפיינים מיוחדים:

- שיתוף פעולה
- משאבים חיצוניים
- אין קישור ישיר למשאב אותו מנהלים

מה כל כך גרוע בעומס? אם יש עומד במערכת, פאקטים לא מגיעים ליעדים, ויש לשלוח אותם בשנית. פעולה זו מחייבת תקורה נוספת, וכן היא מעכבת את המערכת. כתוצאה מכך, כל התקשורת מאיטה.

יתרה מזו, ההשפעה של עומס היא מאוד קיצונית - לא מדובר בתופעה בה קצת עומס גורם לירידה קטנה בביצועים. כאשר פאקט "נופל" (לא מתקבל מחוסר מקום בבאפר של המקבל) ה־node השולח שולח אותה שוב, ובכך מגדיל את העומס על מערכת עמוסה בלאו הכי. כתוצאה מכך, פאקטים נוספים "נופלים". כאשר תהליך זה מתחיל, הוא גורם למערכת לעבור למצב בו רב הפאקטים "נופלים" רב הזמן וכמעט אף מידע לא מגיע ליעדו.

אפשר להשתמש ב־ack־אים על מנת להעריך את התנאים של הרשת מצד אחד, כאשר פאקט עובר ברשת, המשתמש אינו יודע דרך אילו צמתים בדרך הוא עובר. אם השולח קיבל אישור קבלה, הוא רק יודע שהפאקט ששלח הגיע ליעדו (לוקח לו זמן כמובן לקבל את האישור הזה). מצד שני, המקבל לא מקבל הודעה מפורשת על כשלונות (לא מדובר כאן על שגיאות במידע שנפלו בדרך אלא על הגעה ממשית ליעד) - כאשר פאקט לא מגיע ליעדו, לא נשלח nack לשולח. על כן המקבל מסיק שפאקט לא התקבל אם הוא לא קיבל ack מסויים בתוך תקופת זמן.

הבעיה היא איך להחליט מה תהיה תקופת הזמן המתאימה לחכות ל־ack - שכן יכול להיות והוא רק התעכב ברשת והפאקט הגיע ליעדו. השאלה היא איך להבדיל בין שני המקרים. התשובה היא שערך הסף צריך להיות קשור ל־round-trip time, הזמן שלוקח לפאקט לעבור משולח למקבל ובחזרה: ככל שה־RTT גבוה יותר, גם ערך הסף צריך להיות גדול יותר.

איך מחשבים את ערך הסף? עמוד 253 בסיכום של פייטלסון.
הרעיון המרכזי: לא לשלוח יותר פאקטים מהכמות שהרשת יכולה להתמודד איתה. כדי למצוא בדיוק מהי הכמות הזו, כל שולח מתחיל בשליחת פאקט אחד, ומחכה לקבל ack. אם הוא מגיע, שני פאקטים נשלחים, לאחר מכן ארבע, וגו'.
במובנים פרקטיים, שולטים במספר הפאקטים שנשלחים על ידי אלגוריתם חלון, בדיוק כמו flow control. בתשדורת אמיתית, המערכת משתמשת בחלון flow-control וחלון congestion control מינימלי.
האלגוריתם "slow start" פשוט - גודל החלון התחלתי הוא 1. בכל הגעת ack, החלון גודל החלון גדל ב־1.
הבעיה כשמתחילים לאט היא שההתחלה לא באמת איטית, ובשלב מסויים גודל החלון יהיה גדול ממה שהרשת יכולה להתמודד איתו. כשזה קורה, פאקטים "יפלו" ו־ack־אים לא יגיעו לשולח. כאשר זה קורה, השולח נכנס למצב "המנעות מעומס". במצב זה, הוא מנסה להתכנס לגודל חלון אופטימלי, באופן הבא:

- קובע את גודל חלון הסף להיות חצי מגודלו כעת. זה גודל החלון הקודם בו השתמשו, והוא עבד בסדר.
- קבע את גודל החלון להיות 1, והתחל את האלגוריתם "slow start" מחדש. שלב זה נותן למערכת זמן להתאושש מהעומס.
- כאשר גודל החלון מגיע לגודל הסף שנקבע בצעד הראשון, מפסיקים להגדיל את החלון ב־1 לכל ack, ובמקום זאת, מגדילים ב־$\frac{1}{w}$ לכל ack, כאשר w הוא גודל החלון. זה יגרום לגודל החלון לגדול הרבה יותר לאט - למעשה, הגידול יהיה לינארי, יגדל בפאקט אחד לכל RTT. ממשיכים להגדיל משתי סיבות - הראשונה, אולי ערך הסף קטן מידי. שני, התנאים אולי השתנו.

9.4.7 ניתוב

נניח ואנו רוצים לשלוח מייל. כיצד תוכנית המייל יודעת למצוא את המכונה המתאימה? כמה צעדים:

- תרגום כתובת המייל לכתובת IP.
- לנתב את ההודעה מראוטר אחד לשני לאורך הדרך, לפי כתובת ה־IP.

הצעד השני נעשה לפי טבלאות ניתוב, אז שאלה נוספת היא איך ליצור ולתחזק את הטבלאות הללו.

שמות מתרגמים לכתובת IP לפי ה־DNS DNS - Domain Name Server. הרעיון - שימוש בסטרינגים של שמות בעלי משמעות (עבור בני אדם), בדיוק כמו השמות שניתנים לקבצים במערכת קבצים. כמו עם קבצים, נוצרת מערכת היררכית. ההבדל הוא ששמות מיוצגים בסדר הפוך, ומשתמשים בנקודות להפרדה בין המרכיבים.
נביט למשל ב־host: cs.huji.ac.il. il הוא ה־top level domain name, וכולל את כל ה־hosts בדומיין "ישראל". אין הרבה top-level domains, והם כמעט ולא משתנים, כך שאפשרי לתחזק סט של root DNS servers המכירים את כל ה־top-level DNSs. כל המחשבים בעולם חייבים להיות יכולים למצוא את ה־root DNS המקומי, ויכולים לשלוח לו שאילתא עבור הכתובת של il DNS.
בהנתן הכתובת של il DNS, אפשר לשלוח לו שאילתא בדבר הכתובת של ac.il, הכתובת של השרת של ה־hostst האקדמיים בישראל (למעשה, תשלחאליו שאילתא בדבר כל הכתובת, והוא יחזיר את הכתובת הכי ספציפית שהוא מכיר. כמינימום, הוא יחזיר את הכתובת של ac.il). לשרת שימצא תשלח השאילה עבור huji.ac.il, ה־domain server של האוניברסיטה העברית. לבסוף, תוחזר הכתובת של ה־host של מדעי המחשב.
כאשר מוצאים כתובת, היא נשמרת ב־cache לשימוש עתידי. במקרים רבים סט של הודעות ישלח לאותו מחשב מרוחק, ושמירת הכתובת בזיכרון מטמון פוטרת מחיפוש הכתובת שלו בכל פעם.

כתובת IP מזהה את הרשת ואת ה־host בגרסא 4 של IP, כתובות הן באורך 32 ביט, כלומר 4 בייטים. הדרך הנפוצה ביותר לכתוב אותם היא בצורה דצימלית, כאשר הערכים מופרדים ע"י נקודה. החלק הראשון מציין את הרשת, השאר את ה־host ברשת. ה־IP אחראי על ניתוב בתוך רשתות. בהנתן כתובת IP, הוא חייב למצוא כיצד להעביר את המידע לרשת המצויינת בכתובת.

מנתבים צעד צעד ניתוב נעשה באמצעות ראוטרים - מחשבים המחשבים בין רשתות שונות. הראוטרים הללו בדר"כ לא יודעים על כל הרשתות אחרות שניתן להגיע עליהן. במקום זאת, הם יודעים מהו הראוטר הבא ב"כיוון נכון". ההודעה נשלחת לראוטר הזה, שאז חוזר על התהליך צעד נוסף ב"כיוון הנכון". כל צעד כזה הוא קפיצה - hop.
מציאת הכיוון ה"נכון" מתבססת על טבלת ניתוב המכילה את התחיליות, והכתובות המתאימות ל"קפיצה" הבאה. בהנתן כתובת IP, הראוטר מחפש את התחילית הארוכה ביותר המתאימה, ושולח את ההודעה לראוטר הרפיצה הבאה המצויין בשורה זו. בדרך כלל תהיה שורה דיפולטיבית אם אין שום התאמה. שורה זו תעביר את ההודעה לכיוון הראוטרים הגדולים ב־backbone של האינטרנט. לראוטרים אלו יש טבלאות ניתוב גדולות מאוד שאמורות להיות יכולות להתמודד עם כל בקשה.

טבלאות הניתוב נוצרות על ידי אינטראציה בין הראוטרים פרוטוקולי האינטרנט עוצבו במטרה לעמוד בפני התקפה גרעינית, ולא נוטים לכשלונות או פגיעים להתנהגות זדונית. ראוטרים אם כך נוצרים באופן מקומי ללא שליטה מאורגנת.
כאשר ראוטר מופעל, טבלת ניתוב שלו מכילה מידע על השכנים הקרובים אליו ביותר. אז הוא מתחיל בפרוטוקול השולח כל תקופת זמן את כל הדרכים שהוא מכיר על שכניו, עם אינדיקציה של המרחק ליעדים אלו (בקפיצות). כאשר הודעה כזו מגיעה אל כל אחד מהשכנים, טבלת הניתוב המקומית מתעדכנת עם דרכים חדשות או דרכים קצרות יותר ממה שהראוטר ידע קודם לכן.

התרגום לכתובת הפיזית מתרחש בכל רשת טבלאות הניתוב מכילות את כתובות ה־IP של הראוטרים השכנים שאפשר להשתמש בהם בקפיצה הבאה בפעולת ניתוב. אבל החומרה של הרשת לא מזהה כתובות IP - יש לה נוטציה משלה של הכתובות, המגבילה את הגבולות של הרשת. הצעד האחרון בניתוב אם כך הוא התרגום של כתובת ה־IP לכתובת פיזית באותה הרשת. זה בדרך כלל נעשה בעזרת טבלה המכילה את המיפוי של IP לכתובת פיזית. אם הכתובת המבוקשת לא מופיעה בטבלה, נשלחת שאילתת broadcast במטרה למצוא אותה.

אותה הטבלה משמשת למציאת היעד הסופי של ההודעה, ברשת האחרונה בסט.