מערכות הפעלה - 80876

19 ביולי 2012

מרצה: דני דולב
מתרגלים: סיון סבתו, עומר לב.

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. אהבתם? יש עוד!
http://bit.ly/integrali

הערה: סיכום זה מכיל את "זרם התודעה" של המרצה, על כן כדאי ומומלץ להעזר ברשימות אלו יחד עם המצגות הרשמיות של הקורס (שכן לא הספקתי להקליד את כל מה שרשום במצגות גם כן).

תוכן עניינים

I שיעורים **3**

0.1 סקירת החומר . . . 3

1 תהליכים ות'רדים . . . 8
- 1.1 המצב של תהליך . . . 8
- 1.2 ת'רדים - תהליכונים . . . 9
- 1.3 שני סוגי הת'רדים מול תהליכים - סיכום . . . 10
- 1.4 ריבוי תהליכים . . . 10

2 תזמונים . . . 12
- 2.1 זימון online לעומת offline . . . 13
- 2.2 מערכת אינטרקטיבית רגילה . . . 14

3 מקביליות - Concurrency . . . 19
- 3.1 הגדרת התנהגות תוכנית מקבילית . . . 19
- 3.2 בעיית הקוד הקריטי . . . 20
- 3.3 פתרונות לקטע קריטי - ללא תמיכת חומרה . . . 21
- 3.4 פתרונות נתמכי חומרה . . . 22
- 3.5 טרנזאקציות אטומיות . . . 25
- 3.6 דדלוק . . . 26

4 ניהול זיכרון . . . 27
- 4.1 הגדרת הבעיה . . . 27
- 4.2 הקצאת זיכרון - הגדרות והקצאה רציפה . . . 28
- 4.3 *Paging* . . . 29
- 4.4 סגמנטציה . . . 31
- 4.5 זיכרון וירטואלי . . . 31

5 מערכות קבצים - File Systems . . . 35
- 5.1 קובץ . . . 35
- 5.2 Directory . . . 37
- 5.3 מימוש מערכות קבצים . . . 38
- 5.4 עקביות בעת התאוששות מנפילות . . . 42
- 5.5 השפעת שינויי הטכנולוגיה . . . 43
- 5.6 מערכות קבצים מבוזרות . . . 44

6 I/O ותקשורת בין תהליכים . . . 46
- 6.1 מערכות קלט\פלט - I/O . . . 46
- 6.2 Disk Caching . . . 49
- 6.3 תקשורת בין תהליכים (העברת הודעות) . . . 50
- 6.4 Protocol Stacks . . . 51

7 מערכות הפעלה מבוזרות ווירטואליות . . . 56
- 7.1 מבוא למערכות מבוזרות . . . 56
- 7.2 וירטואליזציה . . . 59

II תרגולים **63**

חלק I
שיעורים

0.1 סקירת החומר

זהו קורס מסובך שכן יש הרבה מאוד מערכות הפעלה, והשונויות ביניהן גדולות. עם זאת, יש להכיר את העקרונות הראשיים המופיעים בכל מערכת הפעלה.
מטרת צד של הקורס -להכיר מערכת מחשב בסיסית (למרות השונויות בין המערכות).

0.1.1 חומרים לקורס

המרצה נצמד לרשימות של דרור פייטלסון. עם זאת, הרשימות קצרות ובספרים יש הרבה יותר:

- טננבאום - Modern Operating Systems.
- לתרגילים חומרים ילקחו מהספר:

  The Design of the UNIX Operating System - Bach

- מצגות באתר, אולם לפני השיעור לא תהיה הגרסא האחרונה (משתנה מידי פעם).

0.1.2 מהי מהערכת הפעלה?

15/3/2012

[1]בהגדרה של משפט אחד, מערכת הפעלה היא המתווך בין עולם המשתמש לעולם החומרה.
המטרות שלה:

- לאפשר לבצע את מה שהמשתמש או האפליקציות שרצות רוצות לעשות.
- לעשות את מערכת המחשב עצמה כמה שיותר פשוטה לשימוש.
- במידה ויש יותר ממשתמש אחד\אפליקציה אחת - שהחומרה תבצע את הפעולות בצורה היעילה ביותר.

מעט יותר לעומק - יש אוסף אפליקציות שרצות מעל חומרה. מערכת ההפעלה מחליטה איזו אפליקציה רצה בנקודת זמן נתונה. אפליקציה יכולה לפנות בצורה ישירה לחומרה. אנו מבדילים בין שני סוגים של פקודות לחומרה: פעולות רגילות שהאפליקציה פונה ישירות לחומרה, ופעולות שדורשות רגישות יתר הפועלות אך ורק דרך מערכת ההפעלה (מניעת התנגשויות בין משתמשים למשל).
אינטרפטים - לפי החלטת מערכת הפעלה מפסיקים או ממשיכים, למשל "קריאה מהדיסק נגמרה", הגבלות על CPU, ועוד.
התפיסה של מערכת הפעלה היום היא תפיסה מאוד גנרית - היום עובדים עם מערכת הפעלה וירטואלית - VMware - מרשה להפעיל מספר מערכות הפעלה על מכונה אחת בו זמנית. כל אחת מהן רצה "כאילו היא היחידה בעולם", למרותך שלמעשה הכל פועל מעל אותה מכונה. כלומר, המערכת הרבה יותר מורכבת ממה שאנחנו נכיר במסגרת הקורס. כמו כן, בתוך החומרה עצמה יש הרבה מאוד CPUs שרצים בצורה מקבילה. אנחנו לשם פשטות נניח שלמערכת שלנו יש CPU אחד בלבד, אולם נלמד עקרונות שנוכל ליישם גם על מערכת מרובת CPUs.
מטרת השיעור היום - להכיר אלמנטים שנראה לעומק במהלך הקורס, אז בשיעור זה יהיו קפיצות, ולא נכנס לעומק לאף נושא.

תהליך הזרימה במערכת הפעלה מערכת ההפעלה מבצעת וקובעת לוח זמן לאפליקציה.
דוגמא לרצף של פעולות:
כאשר אפליקציה רצה היא מתבצעת כאילו היא היחידה בעולם, ועובדת ישירות מול ה-CPU - מערכת ההפעלה עצמה לא משחקת תפקיד בזמן הריצה.
נניח והשעון אומר שהאפליקציה רצה יותר מידי זמן (למניעת בעיות שונות). השעון אם כך יפנה למערכת ההפעלה וישלח אינטרפט. מערכת ההפעלה מחליטה אם להפסיק בינתיים את פעילות האפליקציה ולהמשיך עם אפליקציה אחרת.
נניח ומערכת ההפעלה החליטה להפעיל אפליקציה אחרת (context switch). האפליקציה השניה תרוץ זמן מסויים על החומרה, ושוב למערכת ההפעלה לא יהיה שום תפקיד בזמן הריצה. אם האפליקציה מראה רצון

[1]לא נספיק לעבור על כל המצגות, לא נחזור אליהן, לא עקרוני, שכן נטפל בהכל בשבועות הקרובים. נשאר לקורא השקדן.

לקריאה של מידע למשל (דרך הפקודה המתאימה), מערכת ההפעלה מתערבת - מרדימה את האפליקציה השניה, מפעילה את פעולת המידע (I/O) הרצויה, ובוחרת אפליקציה אחרת שתרוץ במקומה (עד שתסתיים פעולה המידע).
במסגרת הקורס ניגע בסיבות של מערכת ההפעלה לעבור בין האפליקציות השונות ובתהליכים השונים הללו.

מה מערכות הפעלה עושות כאמור מטרת מערכת ההפעלה לספק נוחות למשתמש - את המשתמש לא מעניין השימוש במשאבים השונים.
בסביבות שונות, הצרכים משתנים - למשל בטאבלט יש מערכת הפעלה מאוד פשטנית - רק משתמש אחד ומעט מאוד אפליקציות.
במערכות אחרות - הרבה משתנים, הרבה אפליקציות, העדיפויות משתנות, וצריך לחשוב איך להשתמש גם הפעם במשאבים של המחשב. יש מערכות למשל הפועלות כמעט ללא user interface. למשל, מערכות כגון תחנות עבודה הן בעלות משאבים מסויימים אך משתמשות באופן תדיר במשאבים משותפים מסרבירים.
כמו כן בטאבלט או במכשיר סלולרי, ההעדפה היא חיסכון בבטריה לעומת מהירות שימוש. זהו עוד שיקול שמערכת ההפעלה עושה.
היום אפילו בסוויצ'ים ובראוטרים היום מערכת ההפעלה שולטת, ויכולה תחת שיקולים שכאלו להגיע לחסכון אדיר באנרגיה תוך כדי שמירה על התפוקה.
הרבה מערכות (בעיקר במערכות משובצות) מגיבות לעולם חיצוני ואין משתמש המפעיל אותן. למשל חברת מובילאיי הירושלמית מפקחת על הנהיגה, ולנהג אין שליטה. גם זוהי מערכת הפעלה.

הגדרות שונות למערכת הפעלה מערכת ההפעלה כמנהלת משאבים:

- מנהלת את כל המשאבים
- מבצעת את ההחלטות היעילות וההוגנות מבחינת שימוש במשאבים בהנתן בקשות מנוגדות.

מערכת ההפעלה כמערכת שליטה ובקרה:

- שולטת בביצוע תוכניות על מנת להמנע מטעויות, ועל מנת למנוע שימוש לא נכון במחשב.

המטרה היא יותר לפנות להבנה, כי במערכות בהן נשתמש בעתיד אולי נצטרך רק הגדרה אחת.

מרכיבים עיקריים של מערכת הפעלה

- ניהול וזימון תהליכים (process management).
- מערכת ניהול זיכרון (memory management).
- מערכת I/O (קלט פלט - device drivers).
- מערכת קבצים (file system).
- הגנה (protection) ואבטחה (security) - מפני משתמשים זדוניים ולא זדוניים. לא נתעמק בנושא זה (יש קורסים נוספים העוסקים רק בזה).
- תקשורת בין תהליכים לבין מחשבים (networking). גם כאן ניגע בקצה המזלג (שוב יש קורס שלם על הנושא הזה).
- command interpreter (shell), ממשק גרפי. גם כאן ניגע רק מעט ונכיר בעזרת התרגילים.

כל אחד מהרכיבים האלו מנהל גם את כל המשאבים שהרכיב משתמש בהם, וגם מספקת אבסטרקציות (למשל, המשתמש מרגיש שרק אפליקציה אחת רצה), זהו אלמנט שנראה הרבה במהלך הקורס.

מערכת ההפעלה עצמה אינה כמו תוכניות שהכרנו בעבר, אלא מחכה למאורעות ומגיבה אליהן - היא אינה תוכנית "מסודרת" עם קלט ופלט.
היא אחראית על ניהול המשאבים:

- CPU (מי משתמש בו, מתי ולכמה זמן).
- זיכרון לאחסון תוכניות והנתונים שלהם. הזיכרון האמיתי קטן ממה שצריך כדי להריץ את הכל. האפליקציות צריכות להניח שיש מספיק. השאלה היא איך לגרום לאפליקציות לחשוב שיש מספיק כדי שיוכלו לרוץ בצורה נכונה, למרות שמעשית יש מקום פנוי הרבה יותר קטן.

- מקום על הדיסק לקבצים - גישה לדיסק איטית בהרבה, כיצד נחליט מה ישב שם? עוד החלטות של מערכת ההפעלה.

כמו כן, מערכת הפעלה מספקת אבסטרקציה (שכבר דנו בה קודם) ואבסולוציה - מנתקים את הקשר בין אפליקציות שונות כדי למנוע מאפליקציה אחת "לתקוע" אפליקציות אחרות לא קשורות. כלומר, נותנת לכל אפליקציה "הרגשה" שכל המערכת היא שלה.
הזיכרון מופיע בקטעים קטעים לאורך הדיסק (כמו שהכרנו בדיג'י), אבל האפליקציה "חושבת" שכל המידע מופיע ברצף אחד, ומכירה שמות של קבצים במקום כתובות של בלוקים על הדיסק.
אלו הם האתגרים והשאלות של מערכת הפעלה, ואלו ההבנות שנרכוש במהלך הקורס.

כל אחד מהספרים פונה לנושא של מערכות הפעלה מגישות שונות (מערכת הקבצים, ניהול זמנים, ועוד ועוד) - אנו בקורס נלך לפי המערכות הצרכים העיקריים, ונספק פרטים לגבי מבנים ואלגוריתמים רלוונטים.

על הקשר בין החומרה למערכת ההפעלה החומרה מטרתה לעזור למערכת ההפעלה לבצע את התפקידים שלה, זאת בעזרת:

- Bootstrapping
- חיבור למכשירים חיצוניים.
- פסיקות\אינטרפטים.
- הפרדה בין גישה לחומרה במודים שונים.
- שעונים פנימיים.
- תמיכה בפקודות ספציפיות:
  - ניהול זיכרון.
  - סימונים בביטים להפרדה בין פקודות שונות.
  - תרגומים של כתובות.

מערכת ההפעלה מניחה שיש לה תמיכה, בין אם החומרה תומכת, ובין אם למשל ב-VMware שם לא מדובר בתמיכה שהיא ישירות של החומרה.

במצגת 15 מופיעה מערכת מחשב שאנו מכירים. יש מספר מושגים שלכל אחד מהם אנחנו קוראים *device*, ולכל אחד מהם יש *controller* שעוזר לנהל אותו. מערכת ההפעלה עובדת דרך ה-*controller*, לא ישירות מול ה-*device*, ודרכו יודעת כיצד לעבוד.

השלב הראשוני בהתנעת מחשב - בעת הפעלת מתח, המחשב מתחיל לקרוא במקום מסויים מאוד על הדיסק - bootstrap program. זה שלב מאוד רגיש, שייך לעולם החומרה ולא למערכת ההפעלה לדידנו. שלב זה מאתחל את כל האספקטים של המערכת, טוען את הקרנל של מערכת ההפעלה, ומתחילה את הפעולה שלה.
לאחר שמערכת ההפעלה עלתה, ישנם מכשירי קלט-פלט, שצריכים לרוץ יחד עם ה-CPU. ה"בו-זמניות" הזו היא ניצול משאבים, אך יוצרת בעיות אין ספור, אותן נכיר במהלך הקורס.
מערכת ההפעלה יוצרת קשר עם ה-*devices* השונים דרך ה-*controller*. לכל קונטרולר יש באפר, ומערכת ההפעלה דואגת שהמידע יגיע למקום הנכון. ה-I/O הוא מה-*device* לבאפר המקומי של ה-*controller*, משם\לשם מעבירה מערכת ההפעלה את המידה.
כאשר נגמרת העברת האינפורמציה של ה-*device*, הוא מודיע למערכת ההפעלה\CPU שהוא סיים ע"י *interrupt*, ובהתאם לזה מערכת ההפעלה מחליטה על המשך הפעולה. דוגמא לכך ניתן לראות במצגת 18.

0.1.3 פסיקות - interrupts

מערכת ההפעלה צוברת את כל הפסיקות בתוך וקטור של פסיקות, וכאשר היא מקבלת פסיקה השליטה עוברת למערכת ניהול הפסיקות. מערכת ההפעלה בודקת ממי הפסיקה התקבלה, ולפי זה יודעת מה לעשות.
הארכיטקטורה של הפסיקה חייבת לשמות את הכתובת של הפעולה שהופסקה.
פסיקות נכנסות מנוטרלות כאשר פסיקה אחרת מטופלת על מנת למנוע איבוד פסיקות.
תכנון יעיל מעולם הפסיקות הוא מאוד מהותי לפעולה התקינה של מערכת ההפעלה. תכנון לא נכון גורמת למערכת ההפעלה לפעול בצורה מאוד איטית. אנחנו נכיר הרבה מהעקרונות הללו במהלך הקורס.

$trap$ הוא פסיקה שהתוכנה יוצרת, הנגרמת עקב טעות או בקשה של המשתמש.
אחד הבעיות בעת קבלת פסיקה - מה קורה בעת טיפול בפסיקה מתקבלת פסיקה חדשה? זה יכול לתקוע את כל המערכת, שכן ניתן להכנס ללופ אינסופי. לכן הנושא של תכנון פסיקות הוא קריטי.
עוד אפשרות לפסיקה - תהליך רוצה לסמן משהו לתהליך אחר. גם את פסיקות אלו נכיר במהלך הקורס.
גם הקשה על המקלדת יוצרת פסיקה.
פסיקות מגיעות כמעין "הטרדה" של מערכת ההפעלה (מתח). כל פסיקה אפשרית יכול להיות שיותר מגורם אחד יצר אותה. על כן, הגורם רושם את עצמו, ואז יוצר את הפסיקה. המערכת בודקת את הפסיקה, ומחליטה כיצד להמשיך.
ישנן מערכות הפעלה (למשל כרטיס התקשורת) בהן לא על פסיקה יוצרים "הטרדה", אלא כל פסיקת שעון מערכת ההפעלה מקבלת את הפסיקות שנוצרו מאז הפעם האחרונה שהפסיקות נשלחו. לכן מה שאנחנו נלמד לא תמיד תואמת את העולם האמיתי. הפסיקה היא למעשה הדרך להעביר את "שרביט הניצוח" בין האפליקציות השונות.

I/O כאשר מתחילה פעולת I/O, האפליקציה שיצרה את הבקשה או "מחכה ל" יורדת מה-CPU, ומתחיל התהליך של העברת המידע. מערכת ההפעלה מניחה שהאפליקציה יכולה לחכות רק לבקשת I/O אחת, כדי לפשט את העולם גם מבחינת התהליך וגם מבחינת ה-I/O - את מי מעירים ומתי. בהמשך הקורס נראה איך עוקפים את ההנחה הזו.
לחילופין, לאחר שמתחילה פעולת I/O, השליטה יכולה לחזור לאפליקציה מבלי לחכות לסיום ה-I/O, נראה יותר על כך בהמשך.
לעיתים הבאפר מחזיק את המידע ומערכת ההפעלה שואבת ומעתיקה אותו לזיכרון הראשי (למשל ברב כרטיסי התקשורת). יש מושג שהתפתח עם השנים שנקרא DMA - כדי למנוע את המצב של המתנה בעת העברת המידע, לחלק מה-$devices$ (אלו שעובדים יותר מהר או שעובדים עם הרבה מידע), שיטה זו מעבירה את המידע ישירות לזיכרון ללא התערבות ה-CPU, ורק לאחר שכל המידע הגיע לזיכרון מועברת הפסיקה. הבעיה - אנחנו צריכים "לתקוע" את הזיכרון כי אנחנו לא יודעים מתי נשתמש לו.

0.1.4 מבנה זכרון

מערכת ההפעלה אחראית להחליט איזה מידע עובר ממי למי ומתי, בהתאם לצורת העבודה שלנו.

- הזכרון הראשי שלנו - מחולק לדפים, בכל אחד מידע כלשהוא, ומערכת ההפעלה יכולה לפנות לדף כזה או אחר. כאשר מפסיקים להעביר בו מתח המידע הולך לאיבוד (volatile). נכיר זכרונות נוספים בהם זה לא קורה.
- הזכרון המשני הפופלרי ביותר היום - הדיסק[2]. כאן כאשר מפסיקים להעביר מתח המידע אינו הולך לאיבוד. הקבצים עצמם יכולים להיות מפוזרים על פני הדיסק. איך לפזר את הבלוקים על הדיסק בצורה יעילה "היא אומנות בפני עצמה", ונכיר מעט ממנה במהלך הקורס. עוד שיקול - מתי להעביר מידע בין הזכרונות השונים? את כל אלו נכיר במסגרת הקורס.

מערכות האחסון מסודרת לפי הררכיה, לפי התכונות של הזכרונות השונים:

- עלות.
- מהירות.
- מידע הולך לאיבוד כאשר מפסיקים להעביר מתח?

Caching - שמירה זמנית של מידע לזיכרון מהיר. מתי, ואיך?
הירכיית זיכרון מופיעה בשקופית 26 - ככל שעולים, המחיר עולה ואיתו גם המהירות:

- magnetic tapes - מאוד מאוד איטי, בזמנו היה מאוד זול, בזמנו היו שומרים עליהם דברים לשמירת זיכרון לטווח ארוך. פעם היו עובדים איתם ישירות. היום אין בהם הרבה שימוש.
- optical disk - שוב לבאקאפ.
- magnetic disk - הדיסק שאנו מכירים. פרוץ לתקלות.
- electronic disk - מידע לא נמחק ממנו, ומכיוון ואינו מכני פחות פרוץ לתקלות. החסרון שלו - מספר הפעמים שניתן לכתוב עליו מידע מוגבל בשל בלאי.

[2] למרות שהמרצה צופה שזה ישתנה בשנים הקרובות.

- main memory - מוגבל בכמותו במחשב נתון.
- מעליו יש בעצם מספר שכבות (במחשבים מודרנים), במצגת מופיע רק ה־cache בשכבה יחידה.
- ולבסוף - רג'יסטרים.

אם מטפלים במידע ביותר ממקום אחד, צריך לדעת מהו המידע העדכני ואיך עובדים במקרה שכזה. גם על כך נלמד.
Caching - מידע שמועתק מזיכרון שיותר יציב לזמן קצר. יש מעט cache והרבה מידע, ולכן ישנם שיקולים מתי להעלות ל־cache ומה בדיוק.

0.1.5 אלמנטים יסודיים בארכיטקטורת מחשב

במחשב עצמו יש CPU אחד או כמה. ההנחה שלנו בקורס כרגע היא שיש CPU כללי (לא מחשב ייעודי).
במערכות עם יותר מ־CPU אחד (נכיר את כל המושגים הקשורים לזה) נוכל להריץ במקביל הרבה דברים. מחשבים מקבילים עושים דברים שכאלו. כאמור נכיר מודלים שכאלו בהמשך. ההחלטה עם איזו מערכת לעבוד קשורה באיזו פעולות נעסוק הכי הרבה - נכיר את האלמנטים השיקולים השונים ומה עומד מאחוריהם. כאשר העולמות איתם עובדים זרים לחלוטין אפשר להריץ אותם בכל מערכת שהיא.
היום בהרבה מהמחשבים שיש, יש הרבה ליבות. היום החומרה מקדימה את מערכת ההפעלה - כלומר, אין לנו דרך יעילה לנצל הרבה ליבות. היום מתחילות כמה טכנולוגיות בעולם זה. אנחנו נהיה הדור שנצטרך לפתור את זה.

כיצד פועל מחשב מודרני בשקופית 29 מופיעה הארכיטקטורה של וון־ניומן. סכמה של ה־CPU בשקופית 30. הוא מורכב מ:

- ALU - יחידת חישוב מתמטית.
- PSW - processor status word.
- PC - program counter - מצביע לפעולה הבאה.
- SP - stack pointer.
- MEM - פוינטרים לזיכרון.
- רג'יסטרים.

דוגמא לפעולה - שקופית 32.
איך דואגים לכך שאפליקציה לא תפנה לאיזורי זיכרון שאסורים לה (של תוכנות אחרות או של מערכת ההפעלה עצמה)? MEM משגיח, אם יש זליגה היא נעצרת. הרבה נסיונות פריצה למחשב נעשים ע"י נסיון לפרוץ למערכת הזו - לפרוץ לאיזורי זיכרון שונים.
מה ההבדלים בין ארכיטקטורות שונות? האריזה המקובלת (מפוקחים על ידי מערכת אחד) מול המולטיקור (פועלים מול אותו הזכרון - שקופית 35).
clustered systems - לחישובים מאוד מורכבים, הרבה CPUs, אוספים בסוף התהליך את כל התוצאות - הרבה מערכות הפועלות יחדיו.
כיצד מערכת זו עובדת? כל מחשב עובד לבד, יש אפשרות לקשר ביניהם, משותף איזור זכרון, יש אפליקציה שאוספת המידע

במערכת הפעלה אפשר להפעיל מספר מערכות\תהליכים. נכיר את מערכת השיקולים - לאפליקציות שונות יש העדפות שונות, וצריך לראות איך לעבוד עם העדפות אלו.
שיקולי תזמונים - נרצה שהמחשב יתן למשתמש את התחושה שהכל רציף. לתוכניות שונות דרישות שונות - לחיצה על מקשים, שמיעת מוזיקה, ועוד.
תהליך מבחינה קונספטואלית הוא תוכנית או סדרת פקודות שצריכה לרוץ, שמתייחסת למחשב כאילו המחשב שלה למרות שיש הרבה תהליכים. נכיר גם את הת'ראדים.
נעסוק גם בנושא של החלפת המקומות בהם נמצא זיכרון, וכן בנושא של זיכרון וירטואלי.
בעקרון, בזיכרון יש איזורים שונים שמוקצים לתהליכים שרצים כרגע, כל אחד מקבל מעשית הרבה פחות ממה שהוא חושב שיש לו, ומערכת ההפעלה מחליטה עבור כל אחד כמה זיכרון הוא יקבל.
יש אפשרות לסמן פעולה כבעלת עדיפות גבוהה - הביטים מסומנים, ולאחר שמתבצע הביטים משתנים בחזרה. היום יש אפשרות 4 רמות, אבל לרב רק 2 (יוזר מוד מול קרנל מוד), וכן יש הכנה למספר גדול יותר של רמות, כשכרגע לא משתמשים בהם.

1 תהליכים ות'רדים

22/03/2012

הגדרה 1.1 תהליך (פרוסס) הוא אינסטנס דינמי של אפליקציה - אבסטרקציה של "מחשב יחיד".

תהליכים מספקים קונטקסט, המוגדר ע"י:

- מצב ה-CPU (ערכי הרגיסטרים).
- מרחב הכתובות (תכולת הזיכרון).
- הסביבה (כפי שהיא מתבטאת בטבלאות של מערכת ההפעלה).

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

הגדרה 1.2 החלפת התהליך הרץ נקראת context switch - החלפת הקשר.

1.1 המצב של תהליך

1.1.1 מצב 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.

1.1.2 מצב הזיכרון

מורכב מ:

- 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.

1.1.3 שאר המצב של התהליך

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.

1.1.4 מצבי תהליכים

כאשר תהליך מתבצע, הוא משנה מצב - state:

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

דיאגרמת מעברים מופיעה בשקופית 9.

1.2 ת'רדים - תהליכונים

תהליכים מרובי ת'רדים מכילים מספר ת'רדים.
ת'רדים שימושיים בתכנות - למבנה, פונקציונליות, אולם מסובך לשלוט בהם.

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.

דוגמאות לתהליכים בשקופיות $6-15$.
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.
ת'רדים לעומת זאת מאפשרים לכתוב תוכניות סדרתיות עם blocking calls ומצד שני מאפשרים מקביליות.

מי מממש את האבסקטרקציה של הת'רדים? מבחינים בין שני סוגי מימוש עיקריים:

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

יש מערכות הפעלה התומכות בשני סוגי המימוש.

1.2.1 Kernel-Level Threads

ממומשים בקרנל כאוסף של קריאות מערכת. לכל תהליכון יש תא בטבלה של הת'רדים של כל המערכת.

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:

- מקבילות אמיתית בין תהליכונים במערכת רבת מעבדים.
- קל לתמוך במקבילות אמיתית בין CPU ל-I/O.
- preemptive scheduling - מאפשר הפקעה ברמת התהליכון.

לכל ת'רד יש stack משלו ו-descriptor.

1.2.2 User-Level Threads

ממומשים כ-library calls מכל runtime שרץ ב-user space. מערכת ההפעלה אינה מודעת לקיום הת'רדים.
יתרונות:

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

חסרונות:

- לא מאפשרת מקבילות אמיתית (מבחינת ביצועים) עם ריבוי מעבדים.
- אין preemptive scheduling ברמת הת'רד - צריך $yield$ מפורש.
- קשה לאפשר מקבילות בין ה-CPU ל-I/O - blocking system calls 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)?

1.3 שני סוגי הת'רדים מול תהליכים - סיכום

	תהליכים	Kernel Threads	User Threads
שיתוף זיכרון	רק בקריאות מערכת מיוחדות	הכל חוץ ממשתנים לוקליים של פונקציות	
תקורה - overhead	גבוהה - החלפה יקרה בקרנל	בינונית - מהיר בקרנל	נמוכה - ביזאר 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	X	X	✓

1.4 ריבוי תהליכים

יש יותר מתהליך אחד במערכת בזמן נתון. כל מעבד מריץ אחד מהם בכל זמן נתון.
תחילה, קצת טרמינולוגיה.

הגדרה 1.3 מולטיטסקינג - מצב בו מספר תהליכים רצים על מעבד אחד בעזרת time sliceing.

הגדרה 1.4 מולטיפרוגרמינג - מצב בו יש מספר עבודות במערכת (על אותו המעבד, או על מעבדים שונים).

הגדרה 1.5 מולטיפרוססינג - מצב בו נעשה שימוש במספר מעבדים עבור אותה העבודה או המערכת.

1.4.1 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.

1.4.2 המחיר של ריבוי תהליכים

- תקורה - context switching overhead - מזבזים סייקלים על החלפת הקשר.
- 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.

2 תזמונים

29/03/2012

התחלנו בשיעור שעבר בהגדרות השונות כדי להבין מהן המטרות. נראה כי המטרות לעיתים סותרות אחת את השניה, ונראה כיצד לפתור סתירות אלו.
אנו נדון באלגוריתמי זימון offline לעומת online.

הגדרה 2.1 turnaround time - זמן ביצוע כולל. מורכב משני רכיבים:

- זמן ריצה
- זמן המתנה (כמה זמן המשימה ממתינה לביצוע.

אנו נניח כי המשימה פשוטה - צריכה רק CPU, בהמשך היום נרחיב גם לנושא ה-I/O.
לעיתים מעניין אותנו זמן ממוצע.
השליטה שלנו היא רק על כמה זמן ממתינה המשימה.

הגדרה 2.2 Slowdown - כמה זמן תהליך מתעכב עקב ההחלטות של מערכת ההפעלה.

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 2.3 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 הטור.

הגדרה 2.4 utilization יצילות - 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 2.5 fairness הוגנות.

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

הגדרה 2.6 הוגנות חלשה - כל מי שמבקש שירות בסופו של דבר מקבל, כלומר אין הרעבה starvation.

הגדרה 2.7 הוגנות קצת יותר חזקה - לא מחכים זמן לא חסום.

הגדרה 2.8 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 2.9 predictiability יכולת חיזוי - יודעים להעריך כמה זמן משימה תחכה בתור, או משימות מסוג מסויים.

היום לא נעסוק בו הרבה, אבל הוא מאוד חשוב.
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.

2.1 זימון online לעומת offline

הגדרה 2.10 זימון $offline$ - תכנון מראש בידיעה מלאה. לא מציאותי, אבל נותן אינטואיציה ובסיס להשוואה - baseline.

הגדרה 2.11 זימון $online$ - תכנון במהלך הריצה ללא ידיעה מראש.

מערכת רגילה היא שילוב של השניים. נתחיל לטפל בנושא של offline כי קל יותר לטפל בו - היום רב הזימונים הם online.

2.1.1 זימון אופליין

נניח שאנו מקבלים את כל המשימות על ההתחלה. אזי המשימה שלנו היא למעשה לסדר את התהליכים הנתונים בסדר נכון.

- ה־scheduler 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.
- האלגוריתם הפשוט ביותר - 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 התזמון לא השפיע על התפוקה.
- ניתן להראות כי בתנאי הבעיה הנ"ל, SJF הוא הפתרון האופטימלי לכל קלט.
  - אם יש סדרת משימות ידועה, ונסדר אותן בסדר עולה, אם נעצור משימות לא נוכל לשפר דבר, כלומר preemption לא עוזר, רק מגדיל את ה־overhead.
  - נניח בשלילה שיש מקרה בו SJF לא אופטימלי. אזי קיים קלט עבורו קיים תזמון עם זמן המתנה ממוצע מינימלי, בו תהליך כלשהוא מזומן מייד לפני תהליך קצר ממנו. ע"י העברת התהליך הקצר לפני התהליך האורך ממנו, זמן ההמתנה של הקצר מתארך, בסתירה.

2.1.2 זימון Online

עדיין יש משאב יחיד, אולם המשימות לא נתונות אלא מגיעות אונליין, 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 $preemption$.
התאמת הזימון לתנאים המשתנים.
כדי לפצות על המקרה בו תהליך ארוך מאוד רץ ומגיע תהליך קצר יותר תוך כדי - משתמשים ב־$preemption$.

אלגוריתם SRT כאמור המטרה היא להקטין את זמן ההמתנה הממוצע של התהליכים.

ניתן להוכיח שה- SRT מבטיח זמן המתנה ממוצע אופטימלי, בתנאי שמזניחים את התקורה של ה־context switch בשל ה־preemption. אפשר לבנות הרבה מקרים פתולוגיים סותרים במידה ולא מזניחים את התקורה.
הבעיה 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.

אלגוריתם 1 אלגוריתם זימון אונליין - SRT - Shortest Remaining Time

מניחים כי זמן הריצה של תהליך ידוע עם הגעתו (במערכות אמיתיות בד"כ נתון חסם עליון או שערוך של זמן הריצה, אנו נניח בשלב זה שהשערוך שנתון מדוייק). אזי, כאשר מגיע תהליך חדש:

- כצריך להשוותו לתהליכים הישנים. בהנחה שהם מסודרים במערכת כ-SJF - משווים אותו לזמן שנותר לתהליך הנוכחי לרוץ.
- אם זמן הריצה יותר ארוך מהג'וב שכרגע רץ - נכניס אותו לתור.
- אחרת, נפסיק את הג'וב שרץ כרגע ונריץ את החדש - *preemption*.

מחייב ידע מוקדם - כלומר, אם אנחנו יודעים את זמני הריצה. אחרת, יש לשערך את זמני הריצה. זוהי תורה שלמה. אנו נעקוף את הבעיה בדר"כ ע"י חלוקה קבועה. בחלק מהמערכות נעשית למידה תוך כדי פעולה, ואז ניתן להעריך בצורה יותר נכונה.
עם זאת, אלגוריתם זה לא ישים במערכות אינטרקטיביות:

- לא ניתן לדעת מראש את אורך התהליך.
- לא responsive - לא נותן להרבה משתמשים הרגשה שהתהליכים שלהם רצים בו-זמנית. אנו ננסה לשפר את חווית המשתמש.

2.2 מערכת אינטרקטיבית רגילה

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

במערכת טיפוסית תהליכים משתמשים לסירוגין ב-CPU וב-I/O. על כן, התזמון צריך לנסות לייעל את שימוש כל האמצעים במערכת ולא רק ה-CPU.

הגדרה 2.12 קטע CPU בין שתי פעולות I/O נקרא CPU burst.

מהסיבות שציינו מעלה, תמיד נשתמש ב-preemtion.
נניח שגם זמני ההגעה אינם ידועים, וזמן החישוב אינו ידוע.
לפני שנדון בפתרונות, צריך להגדיר מהן המדדים שמעניינים אותנו.

- מה המשמעות של זמן ביצוע כולל במערכת כזו?
  - לא רק זמן ביצוע וזמן המתקנה, נוסף גם זמן לפעולות I/O.
  - לא בשליטת ה-scheduler של ה-CPU.

במקום מדד זה, נמדוד:

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

הגדרה 2.13 זמן תגובה response time - זמן ביצוע כולל של CPU burst (לא תהליך שלם).

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).

אבל גם "זמן תגובה ממוצע לא תמיד מספיק טוב, כי הממוצע לא אומר הרבה", אלא מה התחושה של המשתמש - responsiveness.

תפוקה - בזימון נכון של ה־CPU אפשר לשפר את הנצילות של התקני I/O, ובכך להגדיל את התפוקה שניתן להזרים במערכת.

הגדרה 2.14 תהליכים compute bound הם תהליכים שמנצלים הרבה CPU בין פעולות I/O.

הגדרה 2.15 תהליכים I/O bound הם תהליכים המנצלים מעט CPU בין כל שני קטעי I/O (למשל מעבד תמלילים).

אי אפשר להבטיח תמיד ניצול מקסימלי של compute bound ו־I/O bound, תלוי בתהליכים הספציפיים של המשתמש.

עם השנים "קורה דבר מוזר" - זמן הגישה לדיסק השתפר רק במעט (תהליך מכני), ואילו ה־CPU גדל עשרות מונים מאז שנות ה־70 - זה נכון על הנייר, אבל האמת היא שכמות הזיכרון במחשב גדלה מאוד, והשיפור נובע מכך. מאחר וה־CPU רץ יותר מהר, אפשר להרשות overhead בתנאי שמנצלים נכון גם את ה־I/O. המטרות שלנו משתנות בהתאם לשינוי בעולם - למשל, לפני 40 שנה לא היה מעבד תמלילים, היום צריך להתייחס לתוכנות שונות שהיום בשימוש ולא היו קיימות בזמנו. אם כך, מי שצורך מעט CPU אחרי כל פעולת I/O צריך לקבל עדיפות.

על כן המטרות שלנו הן:

- שיפור תפוקה.
- שיפור responsivness - זמן תגובה.

מהו הקשר בין המדדים?

- 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.
  - ישנן מערכות יעודיות בהן תהליכים כאלה לוקחים הרבה זמן (חישובים ארוכים), אבל אנו לא מתייחסים למערכות כאלו.
- מכיוון וה־CPU אינו ניתן לחלוקה, נחלק את הזמן ליחידות בגודל קבוע - quantum/time slice, כדי להתקרב כמה שיותר לחלוקה של ה־CPU (כפי שמופיע בשקופית 39) למטה. הבעיה היא שכשעושים את זה ה־overhead עולה, לכן מעשית לא נותנים זמנים קטנים מאוד.

2.2.1 אלגוריתם Round Robin

משתמש ב־preemption. תור התהליכים המוכנים - ready, הוא מעגלי.

דוגמאות לריצה - שקופית $5 - 44$.

בחירת גודל ה־quantum - משפיע על התקורה, ולכן יש לבחור גודל אופטימלי. איך עושים זאת?

quantum קטן הוא:

- הוגן (זימון סוציאליסטי).

- מאפשר זמן תגובה טוב - לא "נתקע" בתהליך ארוך.
- התקורה גבוהה.

כמה context switch עולה לנו?

- כיום - סדר גודל של אלפי סייקלים, מספר מיקרו־שניות.
- בתקורה היחסית עולה עם דורות הארכיטקטורה - מערכת הפעלה גדולה, וכן גם כמות הריגסטרים.

אזי, בעת בחירת ה־quantum, צריך לוודא כי:

- ה־context switch חייב להיות קצר ביחס ל־quantum.
- אבל להקפיד שלא יהיה ארוך מידי, כדי לא לפגוע בהוגנות וב־response time.

. ב־unix ההקצבה היא 100msc.
תכונות ה־RR:

- זמן תגובה מהיר.
- זמן המתנה ממוצע גבוה מ־SRT (דוגמא בשקופית 45).
- מניח שכל התהליכים חשובים באותה המידה.

2.2.2 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.

נציע שערוך גודל CPU burst צפוי של תהליך נתון. אפשר לחזות את העתיד מתוך העבר - כלומר, אנו מניחים שהתהליך דומה לעצמו - מה שרצה בעבר ירצה בעתיד. זה לא תמיד נכון, אבל משתמשים בו הרבה.
כיצד משערכים? נותנים משקל מסויים לעבר הקרוב ולעבר הרחוק, וכך משערכים את זמן הריצה הבא בעזרת מונה דועך (aging של מידע) exponential average לגבי התהליך - הנוסחא מופיעה בשקופית 51. במערכות שונות ה־α מקבל ערכים שונים - אם ניתן חשיבות גדולה יותר לעבר הקרוב, נקבל שערוך שגוי אם בפעם האחרונה היתה סטייה מהשימוש הרגיל.
אפשר ליישם את התור לפי השערוך - כדי להקטין את מספר הפעמים שתהליך יופסק, נכניס קודם את התהליכים להם צפוי פחות CPU - זה לא באמת קורה מעשית, וניגע בזה בהמשך.

אבל, קשה לעקוב אחרי גדלי ה־$CPU-bursts$. על כן נבצע שיערוך בשיטה פשוטה יותר:

- מונה שגדל בכל תהליך רץ בכל פסיקת שעון.
- חלוקה ב־2 של כל המונים באופן תקופתי - דעיכה אקפוננציאלית לפי זמן שחלף, לא לפי מספר ה־bursts.
- עדיפות לתהליכים עם ערכי מונה נמוכים יותר.

דוגמא - שקופית 54.
כדי לתת זימון אינטרקטיבי טוב ראינו שני כללי אצבע:

- משתמשים ב־preemption.

- משתמשים ב־RR בין כל התהליכים שמחכים.

שימוש זה מבטיח לנו:

- תפוקה טובה יותר, כי משפר ניצולת של ה־I/O.
- זמן תגובה.

ניתן להשתמש בזימון מבוסס עדיפויות, ולחזות את הצרכים לפי הכלים שלמדנו.

2.2.3 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 - 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. דוגמא - שקופית 57.
אנו צריכים מנגנון שיעזור לנו לנהל עדיפויות שונות, ובכל עדיפות לנהל את התהליכים שמחכים - על כן נחלק את התהליכים לתורים, ולממש מדיניות פרטית בכל תור בנפרד (באחד RR, באחר מנגנון דינמי שונה, וכו').
תהליך של שימוש בעדיפות יכול להיות כזה שרץ רק לפי העדיפות ומסיים את כל התהליכים בעדיפות נמוכה. מדיניות שיוויונית יותר - הקצאה שווה של CPU לכל תור. דוגמא - שקופית 59. 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.

אז, מה באמת מיושם?
כל המנגנונים שראינו! המומחיות של מערכת ההפעלה היא איך לשלב אותן בצורה שתיתן תפוקה טובה. אפשר לראות שתהליכים זהים רצים באופן שונה במערכות הפעלה שונות.
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.
- base מפריד בין עדיפויות סטטיות בין ה־user ל־kernel, ובין סוגי פעולות\שירותים בקרנל.
- cpu_use - נקבל באופן דינמי.
- הקרנל מעדכן עדיפויות באופן הבא:
  - לתהליך שהולך לישון - עדיפות קרנל גבוהה.
  - לתהליך לקראת חזרה ל־user - הורדת עדיפות.
  - דעון מעדכן עדיפויות של כל התהליכים ב־user-mode אחת ל־10msec.

דוגמא - שקופית 64.
לסיכום:

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

3 מקבילות - Concurrency

19/4/2012

שיתוף משאבים בין תהליכים מחייב את התוכניתן להתמודד עם מקבילות - cocurrency. בעבר - שיתוף משאבים התרחש בעיקר בקרנל. היום השימוש בתהליכונים (ת'רדים) נפוץ, ולכן כמעט כל תוכניתן צריך לדעת להתמודד עם הבעיה הזו.
מה גורם לבעיות?

- שיתוף משתנים בין ת'רדים.
- שילוב ת'רדים לביצוע משימה אחת מחייב תיאום וסנכרון.

מתי יש בעיה?

הגדרה 3.1 מירוץ $(race)$ הוא מצב בו זימון הפעולות (מי רץ בדיוק מתי) משפיע על התוצאה הסופית.

"לרב הכל בסדר, אבל לפעמים קורים דברים מוזרים". מאוד קשה לדבג!

3.1 הגדרת התנהגות תוכנית מקבילית

ההגדרה צריכה להיות אבסטרקטית - ללא התחשבות במהירויות המעבדים, זימון תהליכים, וניהול גישה למשתנים משותפים.
בהנתן שיש מספר תהליכים אבסטרקטים המשתפים ביניהם זיכרון, נרצה להגדיר מה התוכנית צריכה לעשות.

הגדרה 3.2 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 3.3 תכונה $(property)$ היא נוסחא בוליאנית (פרדיקט) על ריצה. נגיד כי תוכנית מקיימת תכונה, אם כל ריצה שלה מקיימת את התכונה.

דוגמאות לתכונות:

- תמיד $0 \leq x \leq 4$.
- x לא יורד (מונוטוני עולה).
- בסופו של דבר $x > 1$ (eventually).
- הסופו של דבר התוכנית עוצרת.

לא תכונות:

- x יכול להיות 1.
- יש יותר סיכוי שבסוף הריצה x קטן מ-4 מאשר להיפך.

ישנן שני סוגי תכונות:

הגדרה 3.4 נגיד שתכונה היא תכונת safety אם היא מגדירה משהו נכון תמיד, כלומר, כל מצבי המערכת "בטוחים".

הגדרה 3.5 נגיד שתכונה היא תכונת liveness, אם היא מגדירה משהו שבסופו של דבר מתרחש.

דוגמא - בעיית ה"יותר מידי חלב:

- יש שני תהליכים.
- אם צריך\חסר חלב, מישהו יקנה - liveness.
- אבל אסור שיהיה יותר מידי חלב - safety.

3.2 בעיית הקוד הקריטי

אנו רוצים לאפשר לתהליכים לבצע פועלות המורכבות מיותר מפקודה אחד 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.

דוגמא שראינו לנושא הזה - נגיד שאנחנו מעבירים כסף מחשבון ראשון לחשבון שני, והמערכת נופלת במהלך הפעולה. נרצה שלאחר השחזור המשתמש יראה אחד משני המצבים:

- הכסף ירד מהחשבון הראשון והתעדכן בחשבון השני.
- הכסף לא ירד מהחשבון הראשון ולא התעדכן בחשבון השני.

שני המצבים הללו, גם אם אחד מהם לא מעודכן, הם נכונים. מצב ביניים אפשרי שלא נרצה שאף אחד יוכל לראות - הכסף ירד מהחשבון הראשון אך לא התעדכן בשני.

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. נראה בהרצאה זו (ובתרגול) מימושים.

3.2.1 מודל לבעיה

ישנם תהליכים המבצעים במקביל תוכנית עם המבנה הבא:

```
do{
entry section
critical section
exit section
remainder
}while (true)
```

יש לממש את ה-entry, exit, וכן אסור להניח כלום על קצבי חישוב יחסיים או הזימון - asynchrony.

3.2.2 הגדרת הבעיה - תכונות

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

הגדרה 3.6 Mutual exclusion (מניעה הדדית) - לכל היותר תהליך אחד נמצא בקטע הקריטי בזמן נתון.

הגדרה 3.7 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 3.8 bounded waiting (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.

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

הגדרה 3.9 deadlock הוא מצב בו אף תהליך לא יכול לבצע צעד שיביא להתקדמות.

הגדרה 3.10 livelock הוא מצב בו תהליכים כל הזמן ממשיכים לבצע צעדים, אך הם לא מובילים להתקדמות.

3.3 פתרונות לקטע קריטי - ללא תמיכת חומרה

הפתרונות הללו לא קשורים למבנה מערכות הפעלה, וחלקם לא נכונים. עם זאת, הם מלמדים עקרונות של סנכרון במערכות עם מקביליות, ומדגימים בעיות שעלולות לצוץ. הם חשובים גם בתכנון מערכות הפעלה וגם באופן כללי בתכנות במערכות מרובות תהליכים\ת'רדים.

3.3.1 האלגוריתם של פיטרסון

בהרצאה מובא אלגוריתם זה כפתרון לבעיה בהנתן שני תהליכים. פתרון זה איננו תלוי מערכת הפעלה, שכן אין שום שימוש בה. בתרגול מובאת ההרחבה של אלגוריתם זה ל־N תהליכים.

הרעיון המרכזי - שימוש במערך משותף של flags לציון "רצון" של תהליך להכנס לקטע הקריטי, ומימוש משתנה\מערך 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.

ניתן להוכיח פורמלית כי פתרון זה מקיים את שתי התכונות הרצויות, וכן מתקיימת תכונת ה־bounded wait.

3.3.2 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. דוגמא מובאת במצגת.

3.3.3 פתרון ל־n תהליכים - אלגוריתם המאפיה

הרעיון:

כל תהליך נכנס ו"לוקח מספר" - $ticket[i]$. 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 לקחת גם - שכן פעולת לקיחת מספר איננה אטומית (במצגת - לא בפידיאף - בעיגול נמצא ערך הטיקט של התהליכים, אפשר לראות את ההמחשה של הבעיה).

מתי נקבל הפרה של ה־Mutual exclusion במקרה הזה? מטה מקרה פשוט שאינו מופיע במצגת:

- תהליכים $1, 2$ מבקשים מספר בו זמנית. הם מקבלים את אותו המספר, אך מכיוון וזו לא פעולה אטומית, יתכן ו־2 עדיין לא בדק את הכל ולכן עדיין לא עוכנה ההקצאה עבורו, בעוד 1 ממשיך הלאה לבדיקת המתנה.
- מכיוון והמספר של 2 עדיין לא עודכן, 1 רואה שאין אף אחד אחר שלקח מספר, ונכנס לקטע הקריטי.
- בעוד 1 בקטע הקריטי, 2 ממשיך, רואה ש־1 ביקש מספר, אולם, בגלל הסדר הלקסיקוגרפי שהוחלט עליו, הוא אמור להיות לפני 1 בתור, לכן הוא נכנס גם לקטע הקריטי.

כיצד מתקנים בעיה זו?

לא נהפוך את הפעולה לאטומית, אלא נוסיף סימן לתהליך שבוחר, ואז תהליכים אחרים יוכלו לראות זאת - מממשים עוד מערך 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.

למה 3.11 עבור $i \neq j$, אם P_i בקטע הקריטי C ו־P_j נמצא בהמתנה W או בקטע הקריטי C, אזי:

$$(ticket[i], i) < (ticket[j], j)$$

מלמה זו נובעת ה־Mutual exclusion, שכן אם P_i בקטע הקריטי ו־P_j בהמתנה, אזי מהלמה, כאשר P_j יבדוק את התנאי לכניסה לקטע הקריטי,

$$(ticket[i], i) < (ticket[j], j) = TRUE$$

ולכן הוא ימשיך להמתין עד ש־P_i יסיים ויסמן $ticket[i] = 0$.

אם P_i בקטע הקריטי ו־P_j בכניסה, הוא יקבל מספר גדול מזה של P_i, ולכן ברור שלא יכנס לקטע הקריטי.

הוכחה: נסמן ב־s את נקודת הזמן בה תנאי הלמה מתקיימים. בנקודה זו, $ticket[i], ticket[j] \neq 0$ כי P_i בקטע הקריטי ו־P_j בהמתנה או בקטע הקריטי.

קיימת נקודה $t_1 < s$ בה P_i ממשיך בשלב הראשון של לולאת ההמתנה, דהיינו בנקודה זו P_i מוצא $choosing[j] = false$.

קיימת נקודה $t_1 < t_2 < s$ בה P_i מחליט להמשיך לקטע הקריטי, כלומר הביטוי $(ticket[j] != 0 \;\&\&\; (ticket[j], j) < (ticket[i], i)) = false$.

כעת ישנן שתי אפשרויות:

- $ticket[j]$ נבחר בין t_1 לבין s - בשלב זה P_i כבר בהמתנה, כלומר החישוב של המספר שלו הסתיים. לכן, P_j מתחיל לחשב את המספר שלו רק לאחר שהמספר של P_i כבר נבחר, לכן ינתן לו מספר גבוה יותר, כלומר $(ticket[j], j) > (ticket[i], i)$ בזמן s.
- $ticket[j]$ נבחר לפני t_1 - אזי ב־t_2, כאשר P_i מחליט להמשיך הלאה לקטע הקריטי, $ticket[j] != 0$ (כי הוא כבר חושב קודם לכן), על כן מכיוון ו־P_i מתקדם, $(ticket[j], j) < (ticket[i], i) = false$, כלומר $(ticket[j], j) > (ticket[i], i)$.

■

הגדרה 3.12 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.

3.4 פתרונות נתמכי חומרה

3/5/2012

נמשיך לפתח את היכולות להבטיח מימוש נכון קטע קריטי - כלומר קטע שתהליך אחד מבצע לבדו.

במקביל לעולם התוכנה שאותו הצגנו, ישנן פקודות בסיסיות שאנחנו מניחים שנעשות אטומיות. במחשב האישי הסט הזה מעט שונה.

אם הסט היה מורחב, אזי יש פעולות מאוד מורכבות שאם יעשו בצורה אטומית, מאוד יסבכו את פעילות החומרה.

עדיין על מערכת ההפעלה לתמוך במתכנת ע"מ שיכול ליישם קטע קריטי בצורה נכונה.

למשל, מנגנון חסימת אינטרפט\פסיקה - "יגיע מה שיגיע, אל תפריע לי". הבעיה: אם הפסיקה לאורך יותר מידי זמן, מעכבים את פעילות ה־CPU. יתר על כן, אם חסמנו פסיקה ושכחנו לפתוח אותה, המחשב נתקע. על כן, חיפשו מנגנונים נוספים.

יתרה מכך, נשים לב כי חסימת פסיקות לא מתאימה למולטי קור, כי החסימה היא לקור אחד בלבד.

מבחינתנו, ישנה רשימה של פעולות שאנו מניחים שהן אטומיות. חומרות תומכות בפעולות אטומיות מורכבות יותר.

3.4.1 Test-and-set-lock

הגדרה 3.13 Test-and-set-lock היא פעולה המשנה משתנה ל־true ומחזיר את ערכו הקודם באופן אטומי.

הבדיקה $TSL(lock)$ - משנה הערך ל־true ובודקת את הערך הקודם. אם הערך הקודם היה false אז אפשר להכנס לקטע, וכן הבדיקה עצמה כאמור משנה את הערך של המנעול, כך שאף אחד אחר לא יוכל להכנס, כל עוד התהליך שנמצא בקטע הקריטי לא יצא (רק ביציאה הוא ישנה את הערך ל־false, ורק אז תהליך אחר יוכל להכנס).

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.

3.4.2 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 - אם מוצאים מישהו שמחכה משנים את המיקום בתור, וכמו כן:

- משנים את ה-$lock$ אם זה אנחנו.
- 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.

3.4.3 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:

3.4.4 Semaphore - פתרון נתמך חומרה

המנגנון הבא שהומצא בחומרה הוא semaphore -

הגדרה 3.14 סמפור הוא משתנה המאותחל למספר שלם אי שלילי, המשותף למספר תהליכים, ומוגדרות עליו שתי פעולות שנתמכות ע"י החומרה כך שהן נעשות בצורה אטומית:

- הפחתה - $P(v)$ (מהמילה "בדוק" בהולנדית - כדי לזכור במידה ולא יודעים הולנדית, קל לחשוב על פעולה זו כ"פחות").
- הוספה - $V(v)$ (מהמילה "ועוד" בהולנדית, וכדי לזכור בעברית - "ועוד").

הסמפור 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 - נקרא גם "מנעול". ערך הסמפור במקרה הבינארי יכול להיות רק 0 או 1.
אפשר להשתמש בסמפור להגביל גישה למשאב למספר קבוע של תהליכים. כמו כן אפשר לממש סידור פעולות, ולפתור את בעיית הקוראים-כותבים וה-producer-consumer.

בעיית הקוראים־כותבים כשמישהו כותב, צריך לעשות זאת בלעדית (כי משנה את מבנה הנתונים), קוראים יכולים לקרוא רבים בו זמנית.
תחילה , יש לשים lock על הכתיבה. גם הקוראים צריכים לבדוק אותו, כי אם מישהו כותב, אם לא יכולים לקרוא.
כדי להרשות לכמה קוראים לקרוא בו זמנית, יש ליישם מנגנון יותר מסובך ממנעול אחד: מממשים counter למספר הקוראים, ודגל קריאה. כאשר הקורא האחרון מסיים, הוא מכבה את מנעול הקריאה. כדי למנוע הרעבה של כותבים, יש לממש פתרון מעט מסובך יותר - יותר על כך בתרגול.

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'.
בשקופית 89: חסימה בקטעים קריטיים של ספירת ממתינים.
ביצוע אטומי של שתי הפעולות מובטח בעזרת:

- חסימות פסיקות
- אלגוריתם שמבצע busy wait.

בספריות הממשמות מנעולים כיום, נוטים לשלב בין שני המנגנונים להמתנה: תחילה מנסים לתפוס מנעול ברמת המשתמש (busy wait), ואם אחרי מספר פעמים לא הצליח להתעורר, מממשים 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.

3.4.5 מוניטורים

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

הגדרה 3.15 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 התי'רדים.
- לא יעיל במקביליות, בגלל החסימה.

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

3.5 טרנזאקציות אטומיות

3.5.1 מוטיבציה והגדרה

ראינו את הסמפורים והמוניטורים. לסמפור רצינו אבסטרקציה גדולה יותר - כך הגענו למוניטור. אולם במוניטור כל גישה לאובייקט היא קטע קריטי אחד גדול - אין מקביליות אפילו בין שני קוראים.
מה היינו רוצים? גרעיניות - נרצה נעילה ברמה קטנה יותר - מניעת גישה משותפת למשאב רק כאשר יש קונפליקט אמיתי.
הבעיה - שוב צרות:

- 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 3.16 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.

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

3.5.2 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. נשים לב שאסור לבצע פעולה שלא ניתן לבטל - למשל I/O.

3.5.3 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

3.5.4 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).

3.6 דדלוק

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

הגדרה 3.17 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.

3.6.1 תנאים 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.
מה עושים?

- 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.
    - משחררים משאבים שכבר מוחזקים אם לא מקבלים משאבים נוספים שמבקשים.
  - 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.

עוד בנושא זה - בקורס מסדי נתונים.

4 ניהול זיכרון

10/5/2012

הזיכרון הוא משאב חשוב. כאשר המעבד מריץ תוכנית, הוא ניגש לזיכרון לצורך ביצוע פעולות מיליארדי פעמים בשניה. על כן, גישה מהירה לזיכרון קריטית לביצועים.
הזיכרון הפיזי בנוי בצורה היררכית, כי זיכרון מהיר נחוץ לביצועים, אבל צריך גם זיכרון גדול וזול. עם השנים, הזיכרון גדל ומוזל, אבל צריכת הזיכרון ע"י התוכנה גדלה (כמעט) באותה המידה.

4.1 הגדרת הבעיה

החומרה ומ"ה מסתירות מאיתנו את ההיררכיה ומספקות אסטרקציה של הזיכרון - מרחב כתובות וירטואלי.
לכל תהליך יש מרחב כתובות וייטואלי משלו - כתובת 1000 של תהליך אחד מכילה מידע שונה מכתובת 1000 של תהליך אחר - ממופה למקום אחר בזיכרון הפיזי.
מערכת ההפעלה ממפה את מרחבי הכתובות (הוירטואלים) של התהליכים לזכרון הפיזי.
כתובות בתכנית הן וירטואליות, למשל:

- הערך שמחזיק מצביע - בפניה למצביע $*ptr$ ניגשים לכתובת הוירטואלית שהמצביע מחזיק.
- הכתובת של המשתנה - בגישה למשתנה foo ניגשים לכתובת הוירטואלית $\&foo$.
- כתובות בקוד - $jmp\ r$ מדלג לכתובת הוירטואלית ש-r מחזיק.

התוכניות צורכות בדר"כ הרבה פחות זיכרון ממרחב כתובות טיפוסי היום. לא כל הכתובות מוקצות - חלק מוקצות בזמן קומפילציה (כתובות סטטיות), אחרות מוקצות בזמן ריצה.
דוגמא למרחב כתובת של 32 ביט - בשקופית 9: ג'יגה שמור לקנרל, בשביל המשתמש - מיקום ה-data והקוד (טקסט) נקבע בזמן קומפילציה. הקצאת זיכרון דינמית מתוך ה-heap, ובין ה-heap ל-stack יש כתובות שאינן בשימוש.

4.1.1 כתובות וירטואליות לעומת פיזיות

לכל תהליך יש מרחב כתובות וירטואליות.
נשים לב כי אם שני תהליכים מריצים את אותה התוכנית, אזי לכל משתנה 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.
בזיכרון הראשי - RAM - יש כתובות פיזיות. כתובת פיזית מכילה ערך אחד ויחיד בזמן נתון. משתנים של תהליכים שונים נמצאים במקומות שונים בזיכרון הפיזי.

4.2 הקצאת זיכרון - הגדרות והקצאה רציפה

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 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. הוא משתמש ברגיסטרים ובטבלאות שמערכת ההפעלה מעדכנת. דוגמא - שקופית 14.

4.2.1 הקצאה רציפה

בשיטה זו, כל מרחב הכתובות (הזיכרון הוירטואלי) של תהליך יושב באופן רציף בזיכרון הפיזי.
כל תהליך במערכת נמצא בשלמותו בזיכרון הראשי. אם אין מקום בזיכרון - $swap$: הורדה של תהליכים שלמים לדיסק.
מימוש - דרך ה-MMU. לא משתמשים בשיטה זו היום.

כיצד ממשים?
הזיכרון מחולק לשטחים שהוקצו ול"חורים. השאיפה- למלא את החורים בזיכרון בתהליכים בגודל המתאים.
כאשר תהליך מזומן לריצה, מנהל הזיכרון מחפש "חור" הגדול מגודל התהליך, והמקצה בו זיכרון בגודל הנדרש.
בסיום התהליך - הזיכרון משוחרר, ומתאחד עם חור שכן, אם יש כזה.
מערכת ההפעלה בודקת אם קיימים תהליכים הממתינים להיכנס לזיכרון אשר יש בשבילם חור מספיק גדול.
הבעיות:

הגדרה 4.1 external fragmantation - פרמנטציה 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 4.2 internal fragmantation - פרגמנטציה פנימית מתרחשת כאשר יש שטחים לא מנוצלים בתוך המחיצות.

ניתוח סטטיסטי מראה שבכל השיטות שהוצעו להקצאת זיכרון רציפה, יש הרבה פרגמנטציות - כשליש מהשטח לא מוקצה.
חוק 50% - בממוצע, הזיכרוש לא ניתן לניצול בחורים הוא מחצית מגודל הזיכרון המוקצה לתהליכים.

בעיה נוספת - גודל משתנה של תהליך - תהליכים גדלים בזמן ריצה:

- הקצאה דינמית של זיכרון משנה את גודל התהליך.
- הגדלת ה-stack (קריאות לפונקציות).

לעיתים משאירים מקום לתהליכים לגדול בין מחיצות, ולפעמים בתוך מחיצה. אם זה לא מספיק, ואין מקום בתוך\ליד מחיצה, יש להזיז את התהליך כולו לחור אחר - וזה יקר.
אם סך כל המקום בזיכרון לא מספיק - מבצעים $swap$ - מעבירים תהליך שלם לדיסק. אם האיזור של ה-$swap$ מלא - הורגים תהליך.

לסיכום יתרונות:

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

חסרונות:

- פרגמנטציה גבוהה (שליש).
- קשה לתמוך בגודל משתנה.
- כל הזיכרון של תהליך חייב להיות בזיכרון - לכן קשה לתמוך בתהליכים גדולים, וכמו גן מגביל את מספר התהליכים בזיכרון (במצב ready).
- swap יקר.

כיצד פותרים?

4.3 *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). מחלקים את הזיכרון הלוקי לבלוקים באותו גודל בשם pages - דפים.
- לא בהכרח כל הדפים של תהליך רץ נמצאים בזיכרון - demand paging: הבאת דפים מהדיסק לפי צורך.
- כתובת וירטואלית 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.

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 בהצעת 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.

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.

דוגמאות - שקופית $9 - 28$.
הטבלאות מאוד גדולות, מה שגורם לבעיות:

- לא ניתן לשמור את הטבלאות בחומרה ייעודית, 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:

4.3.1 TLB - שימוש בזיכרון מטמון

17/5/2012

שקופית 38 - דרך לטפל ב־overhead של ניהול הזיכרון -
בהנחה שהלוקליות נשמרת (כפי שאנו יודעים שקורה לרב), בונים זיכרון מטמון, 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.
כאשר ה־cpu מבקש לפנות לכתובת מסויימת, מפעילים אם כך 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 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.

נמשיך לבעיה השניה - בזבוז מקום:

4.3.2 Multi-Level Paging

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־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 בדיסק, צריך לייבא אותה, ואז לייבא את המידע הדרוש.
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.

פתרון לבעיה השלישית:

4.3.3 Inverted Page Table

הרעיון: במקום לחלק לדפים אבסטרקטיים ולשמור לינקים, שומרים את המקום האמיתי. כלומר, שומרים טבלה יחידה 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.

היינו רוצים דרך יעילה למצוא איפה האינפורמציה בטבלה. נשתמש בשיטה היעילה ביותר שאנו מכירים - $hash\ table$ לחיפוש כניסה בטבלת הדפים ההפוכה.
נשמור בטבלת $Hash$ את מה שנמצא בזיכרון הראשי. הגודל ידוע, ולכן אין חשש שנעמיס עליה, כלומר אפשר לקחת טבלה גדולה מטבלת הדפים.
כיצד זה עובד? שקופית 52.

4.3.4 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?

4.4 סגמנטציה

בשתי השיטות הקודמות שראינו דבר אחד לא מטופל - כאשר מסתכלים על מרחב הכתובות של התהליך, יש את הקוד שאסור לדרוס, המשתנים, וכו'. לכל אחד ממרחבי הזיכרון הללו, היינו רוצים לתת מגבלות שונות לגבי קריאה וכתיבה. הבעיה החלוקה לדפים: יכול להכיל קטעים שצריכים מגבלות שונות.
היינו רוצים "משהו" שיתן לנו את היכולת להגדיר מרחבים מגדלים שונים. אפשר להגדיר סגמנטים (ראינו בשיעור הקודם), אבל הם לאו דווקא עובדים עם השיטות שראינו לחלוקה לדפים, זאת מכיוון והם מגדלים שונים. אם נרצה להשתמש באותה הצורה נקבל פרגמנטציה - בזבוז גדול של מקום.
אזי, משלבים בין השניים:
מחלקים הזיכרון לסגמנטים - חלקים שונים המתאימים לוגית לתוכנית: תוכנית ראשית, פונקציה, מתודה, אובייקט...
מסתכלים על הביטים שמייצגים את הכתובת הזיכרון לא כמשהו עוקב, אלא מסתכלים על חלק מהם כמייצגים את מספר הסגמנט, וחלק מייצגים את המיקום בתוך כל סגמנט - כלומר, כתובת לוגית מחולקת לשני חלקים: מספר הסגמנט, והאופסט בתוך הסגמנט. המיפוי נעשה על ידי טבלת סגמנטים בחומרה.
טבלת הסגמנט ממפה את הכתובת הפיזית לשני חלקים - כל שורה בטבלה מכילה:

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

מגדירים סגמנטים, ולכל סגמנט טבלת דפים. הרעיון הוצא כבר ב־1972, אך חזר לשימוש רק לאחרונה.

4.4.1 שילוב של סגמנטציה ו־Paging

על מנת למנוע פרגמנטציה חיצונית ועדיין לשמר סגמנטים לוגיים, ניתן לחלק סגמנטים לדפים ולהשתמש ב־paging:
לכל תהליך טבלת סגמנטים, לכל סגמנט טבלת דפים. מבנה כתובות אפשרי יכיל שלושה חלקים: הסגמנט, הדף והאופסט.
באינטל עושים את זה קצת אחרת - שקופית 58: שימוש בסלקטור המצביע כניסה בטבלת הסגמנטים, ושתי רמות של טבלאות דפים.

4.5 זיכרון וירטואלי

4.5.1 עוד על demand paging

תזכורת: לא כל הדפים נמצאים בזיכרון בזמן ריצת התהליך. כשתוכנית מתחילה לרוץ, מ"ה מנחשת אילו דפים יהיו בשימוש וטוענת רק אותם לזיכרוץ על מנת להבדיל בין דפים בזיכרון לאילו שאינם בזיכרון (או לא מוקצים), לכל דף יש valid bit בטבלת הדפים.
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.

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

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.
במערכות מולטי-קור, יש מאגר מסגרות משותף לכל המעבדים, ובנוסף, לכל מעבד יש מסגרות פנויות משלו. בכל מעבד (או ליבה) רץ עותק של הקרנל - מקצה ומשחרר זיכרון מהמאגר המקומי.
אם כמות המסגרות הפנויות של המעבד יורדת מתחת ל"קו אדום", מקצים עוד מסגרות ממאגר משותף. אם כמות המסגרות הפנויות של המעבד עולה מעל לקו העליון, משחררים מסגרות למאגר המשותף.
מדוע מנהלים מסגרות פנויות לכל מעבד? המאגר הוא משאב משותף, גישות אליו מחייבות סינכרון - כך רב ההקצאות והשחרורים מתבצעים מול המאגר המקומי, ללא צורך בסנכרון. הזיכרון הפיזי עצמו מחולק בין המאבדים - נותנים לכל ליבה אזור מסויים שתהיה אחראית עליו (אם הכל היה משותף לכל הליבות, היה מתקבל overhead גדול).

4.5.2 אלגוריתם החלפת דפים

נחזור ל-cpu יחיד. המטרה שלנו: למצוא דרך יעילה להחליט מה כדאי "לזרוק" מהזיכרון. אלגוריתם החלפת דפים - Page Replacement Algorithm מחליט איזה דף לזרוק.
זמן הגישה לזיכרון גדל בכל page fault. לכן, התפוקה תלויה ביעול הדרך "לזרוק" דפים. ניתן לבצע context switch בזמן page fault לשיפור הביצועים - אם יש תהליכים אחרים שיכולים לרוץ בינתיים, התפוקה לא בהכרח נפגעת. אולם אם קצב ה-page faults עולה, יש פגיעה.
נמדוד את הביצועים של demand paging ביחס לזמן הגישה.
סימונים:
ma - זמן גישה לזיכרון פיזי (לוקח בחשבון שימוש ב-TLB).
p- ההסתברות ל-page fault (כלומר ההסתברות שהדף שאנו מחפשים אינו בזיכרון).
pft - זמן שירות ל-page fault (כלומר הזמן להביא את הדף מהדיסק).
אזי, זמן גישה אפקטיבי (המדד שמעניין אותנו) מחושב כך:

$$act = ma + p \cdot pft$$

מכיוון ו-$pft >> ma$, האינטרס שלנו הוא להקטין את p.
מהדוגמא בשקופית 14, רואים חשוב שגם הזמינות של הדפים תהיה גדולה.
כאמור, אלגוריתם החלפת הדפים (Page Replacement Algorithm) הוא אלגוריתם של 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 - זמן 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, 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.
דוגמא לריצת האלגוריתם האופטימלי - שקופית 20.
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:

4.5.3 אלגוריתם $FIFO$

הקריטריון: בוחרים בדף הוותיק ביותר להחלפה. ממומש בעזרת מחסנית. דוגמא - שקופית 22.
האינטואיציה אומרת לנו כי הוספת מסגרות אמורה להקטין את מספר ההחטאות. אולם מכיוון והאלגוריתם לא יודע מה היה קודם, כאשר מגדילים את מספר הפיריימים, הוא ירוץ פחות טוב. זוהי אנומליה לא טבעית - Belady's Anomaly - דוגמא בשקופית 23. לאלגוריתם זה יש כמה כאלה. הוא כאמור כלל לא מוצלח.
אלגוריתם מוצלח יהיה זה שידע איזה דף שהכי הרבה זמן לא ניגשו אליו, בגלל עיקריון הלוקליות, סביר שלא יגשו אליו בזמן הקרוב.

4.5.4 אלגוריתם Least Recently Used

ניבוי העתיד בעזרת העבר הקרוב.
אין לנו את המידע הדרוש, אבל במידה והיה לנו, נרצה להגדיר כיצד אלגוריתם זה יעבוד (שקופית 24).
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.

4.5.5 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), מזיזים אותו לסוף התור, ומאפסים את הביט (הביט מאופס ע"י הדיימון ת'רד). דוגמא - שקופית 27.
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). גם אין לנו הבחנה טובה בין דפים בשימוש תדיר לדפים פחות חשובים.

4.5.6 2-Handed Clock

פתרון לחולשה שראינו בפתרון הקודם: 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. דוגמא - שקופית 30-3.
זהו פתרון די חדש, שהגיעו כאשר הזיכרון גדל מאוד.

4.5.7 Enhanced Second-Chance

כעת יש לנו כמה ביטים בזיכרון[3]:

- Valid - האם הדף בזיכרון. מעודכן ע"י מ"ה בלבד.
- Modified/dirty - האם הדף השתנה מאז כתיבתו האחרונה לדיסק. מעודכן על ידי החומרה אוטומטית.
  - מאותחל ל-0 כאשר הדף נטען.
  - מודלק בכל כתיבה לדיסק.
  - מכובה כאשר הדף נכתב לדיסק.

[3] יש גם ביט נוסף בו דנו בשבוע שעבר, אבל הוא לא קשור למנגנון שדנו בו, ועל כן לא נזכיר אותו בזמן זה.

- Referenced - האם ניגשו לדף לאחרונה. מודלק בכל גישה ע"י החומרה, מכובה כאשר המחוג עובר אותו (מאופס ע"י הדיימון ת'רד).

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...
דוגמאות המשוות בין ארבעת האלגוריתמים שראינו - שקופית 37.

4.5.8 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

4.5.9 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 תירד בצורה דראסטית - אז מתחיל thrashing: כמעט כל פעולה דורשת גישה לזיכרון.

הגדרה 4.3 מערכת נמצאת במצב *thrashing* אם היא מבלה יותר זמן בניהול (context switch, paging) מאשר בעבודה (הרצת תהליכים).

ה"אומנות": למצוא כמה תהליכים אפשר להריץ מבלי להגיע ל-thrashing.
workset - לאילו דפים תהליך פונה בזמן מסויים. זה משתנה מתהליך לתהליך. אם נלמד את ה-working sets נדע כמה תהליכים נוכל להריץ בצורה טובה. בגלל העלות הגבוהה של page fault, חשוב להסתכל על טווח זמן גדול. ה-LRU מטרתו לתת קירוב טוב ל-Working Set.
אם "נתקעים", כלומר אי אפשר לתת שירות לתהליכים שרצים, מעבירים תהליכים שלמים לדיסק - swapping, ושינוי מצבם ל-swapped, ואז שאר התהליכים יכולים להמשיך לרוץ.

5 מערכות קבצים - File Systems

24/5/2012

נדון בקובץ כמושג אבסטרקטי. נדון בדרך בה הקבצים מאוחסנים בדיסק.
מערכת הקבצים היא אבסטרקציה שמערכת ההפעלה מספקת. התפקיד - יצירת ישות שיש לה נוכחות מתמשכת, או לאותה אפליקציה שתעלה בפעם הבאה, או לאפליקציות אחרות.
אבסטרקציות עיקריות:

- קובץ.
- מדריך *directory* (תיקיה\מחיצה folder ב־windows).

אנו נדון בשני אספקטים - באבסטרקציה, ובמימוש.
נזכר בהיררכיית הזיכרון שראינו אינספור פעמים בעבר. כאן אנו דנים ב- main memory כלפי מטה, בעיקר על הדיסקים. יש רמות נוספות של אחסון מידע - בעזרת off-line storage. יש אפילו רמה נוספת - גיבויים רחוקים שלא ניגשים אליהם באופן שותף.
היום מחליפה את הרמה השלישית רמת ה־NAS/SAN.
ההבדל בין הזיכרון הראשי לבין הדיסק (שקופית 7):

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)

קצת מספרים - כמה מידע ניתן לשמור היום - שקופית 8.
מספר הנוירונים במוח קטן מטרבייט - המספרים בלתי נתפסים! על כן יש ליעל את הדרכים לגשת למידע.

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.

דרישות

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

5.1 קובץ

הגדרה 5.1 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.

האבסטרקציה מבנה הוא 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 ביוניקס. 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.

מבחינת המשתמש כל המידע הזה יושב יחדיו, אולם זו אבסטרקציה - יכול להיות והמידע מחולק על פני מספר בלוקים.
פעולות על קובץ (שקופית 14):

- יצירה
- מחיקה
- פתיחה
- קריאה
- כתיבה
- קריאת attributes ושינויין.
- rename

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.
ביוניקס - 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.

5.1.1 Memory mapped files

הרעיון: היינו רוצים להסתכל על קובץ כוקטור ענק. שיטה זו ממפה את קובץ או חלק ממנו לאיזור רציף בזיכרון הוירטואלי - ואז עובדים עליו כוקטור מאוד גדול. 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. דוגמא - שקופית 18.

- 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.

5.2 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.

5.2.1 link

אסור למשתמש רגיל להוסיף לינק ל־directory.

מוסיף כניסה ל־directory עבור קובץ קיים. יש לקובץ attribute המציין כמה שמות שונים הוא קיבל - 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, אם הקובץ ימחק, הפוינטר לא ימצא את הקובץ - זוהי ישות זמנית (זו מחיקה אמיתית, בניגוד ללינק רגיל).

5.2.2 פעולות

- mkdir - יוצר מחיצה ריקה.
- rmdir - מוחק מחיצה (מתבצע רק אם היא ריקה).
- closedir/readdir/opendir (לא system calls).
- chdir.
- mount/unmount - צרף מערכת קבצים.

במערכות הפעלה שונות יש פעולות נוספות.

5.2.3 Access control

יותר מאפליקציה אחת יכולה לגשת לישות קובץ. נרצה לבטא את הרצונות שלנו כיוצרי הקובץ לגבי הגישות, ונרצה כי מ"ה תעזור לנו לממש את הרצונות האלו. כמות התיאורים השונים היא עצומה, ואילו היינו רוצים לממש את כולם, הגישה לקובץ היתה בלתי אפשרית. על כן צריך דרך יותר פשטנית.

הגדרה 5.2 Acces Control List (ACL) היא רשימה של משתמשים וסוגי גישה המותרים לכל אחד מהם.

כיצד מנהלים את המידע הזה? לא ישים להחזיק רשימה מלאה, צריך ייצוג תמציתי.
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. במערכות מיוחדות יש ניהול של רמות של הסיסמא (ברמת הרשאות יוזר - סוגי סמאות שונים). לא ניגע בנושא הזה הרבה.

5.2.4 Consistency Semantics

חשוב להגדיר מה התוצאה מכך שכמה משתמשים מעדכנים קובץ במקביל.
אם נסתכל על סדרת פעולות של משתמש 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 להחזיר 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.

האחריות שלנו כמתכנים להבטיח קונסיסטנטיות.
הסמנטיקה הזו נתמכת למשל במערכת קבצים AFS שפותחה מזמן ע"י IBM. המטרה היתה פיתוח מערכת קבצים לסביבה מבוזרת. כל משתמש שפותח קובץ מקבל בלעדיות עליו. מערכת זו מבטיחה קונסיסטנטיות, אך חלשה בבו-זמניות.

5.3 מימוש מערכות קבצים

נושאים עיקריים:

- הקצאה\מיפוי של שטיח דיסק (בלוקים) לקבצים. בדיונים הקודמים כשלא היה לנו מקום זרקנו לדיסק, כאן זו הרמה האחרונה, אז אין לאן "לזרוק".
- תמיכה בפעולות על הקבצים באופן יעיל.
- ניהול שטח דיסק פנוי, כדי להשתמש בהם בצורה יעילה במידה ויש דרישה.
- ישנם נושאים נוספים בהם נטפל בהרצאה הבאה.

הדיסק עצמו מחולק לבלוקים, כאשר בלוק הוא יחידה לוגית של העברת מידע בין דיסק לזיכרון. לא ניתן לקרוא מהדיסק ביחידות קטנות מבלוק. דיסק יכול לעבוד עם בלוקים במספר גדלים.
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 5.3 *volume* הוא יחידה אבסטרקטית שיכול להיות פחות מדיסק, או כמה דיסקים. 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.

5.3.1 הקצאה רציפה

הקצאת בלוקים רציפים עבור קובץ. יש לשמור 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).

5.3.2 $Extent - based$

extent - הקצאה רציפה של בלוקים בדיסק.
לבלוק מוקצים מספר extents שיכולים להיות בגדלים שונים, 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 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.
במקרה בו האקסטנט הוא בלוק בודד - במקרה זה בלוקים של קובץ פזורים בצורה כלשהיא על הדיסק.
במקרה זה הקריאה הסתדרתית 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.

5.3.3 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, לא מאפשר גישה אקראית (seek).
פתרון - הוצאת המצביעים מחוץ לבלוקים:

5.3.4 File AllocationTable (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 שומרת המצביעים, שורה לכל בלוק בדיסק (מה-directory מקשר 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, 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)ץ
- חסרונות:
  - טבלה גדולה - מצביע לכל בלוק.

לכן היום בדיסקים גדולים לא משתמשים ב-FAT. יש פיתוחים שונים לשיטה הזו - FAT-32 פותר את הבעיה אבל עם טבלאות ענק, בהן משתמשים היום.
עוד שיטה:

5.3.5 Indexed allocation

עבור כל קובץ, שומר 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 הולכים מביאים את המידע של האינדקס הבא מהדיסק.
בכמות קטנה של אינדקסים, בסבירות גבוהה מספקים את הצרכים של אפליקציה בנקודת זמן נתונה.

5.3.6 I-Nodes

השיטה המקובלת היום ביוניקס:
עבור כל קובץ שמור מספר (קטן) של מצביעים ישירים בטבלת *inode* - נקראת לזיכרון בביצוע *open*.
אחת הכניסות בטבלה מחזיקה מצביע לבלוק בדיסק המכיל כתובות בלוקים (single indirect).
ב-unix קלאסי (שקופית 47): 13 כניסות, מהן 10 ישירות. מהשלוש שנותרו, 1 לא ישירה, שניה לא ישירה כפולה, ושלישית - לא ישירה משולשת. בלוק מצביעים מכיל N מצביעים, כאשר $N = \frac{block\ size}{pointer\ size}$.

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.

על כן הקבצים איתם עובדים מכתיבים את ההקצאות. צורת השימוש משתנה עם הזמן, ואיתה גם הצרכים של המשתמשים.

5.3.7 NTFS

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 נשמים ב־MFT, וגדולים נשמרים ב־extents בדיסק, עם מצביעים ב־MFT.

במקרה זה מדריך מכיל אטריביוטים מה־MFT של הקבצים בו.

יש תמיכה ב־compression. כמו כן התאוששות מנפילות בעזרת מנגנון טראנזקציות (meta-data בלבד).

5.3.8 ניהול שטח דיסק פנוי

31/5/2012

כדי שנוכל בצורה יעילה לכתוב ולהוריד קבצים לדיסק, יש צורך בשטח פנוי, אחרת אין לנו איפה לשמור את האינפורמציה בצורה אמינה למשך זמן, כי הזיכרון הראשי בזמן כיבוי מערכת ההפעלה נמחק, ולכן אי אפשר לסמוך על מידע ששמור שם[4].

לצורך זה, 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.

ב־FAT כאשר הבלוק פנוי יש 0 בטבלה, כלומר קל לזהות אם בלוק פנוי.

כאשר לא עובדים ב־FAT, צריך דרך לזהות איזה בלוק בדיסק תפוס ואיזה פנוי. טכניקה אחת שבה השתמשו בעבר (היום פחות) - שומרים לכל בלוק ביט המציין אם הבלוק פנוי או תפוס. בעת הדלקת המחשב מעלים את הטבלה הזו לזיכרון ועובדים איתו, יש לוודא תאימות בין העותק בזיכרון הראשי ובין זה על הדיסק.

טכניקה אחרת - לקחת חלק מהבלוקים בדיסק ולשריין אותם כמצביעים לבלוקים הפנויים בדיסק (שקופית 8) - רשימה משורשרת של מצביעים לבלוקים פנויים. כאשר בלוק נוסף מתפנה משרשרים אותו, כאשר נתפס מוחקים אותו. כך יש דרך דינמית לניהול הבלוקים הריקים. זוהי הדרך המקובלת היום, למעט דרך שלישית שנלמד לקראת סוף השיעור היום.

5.3.9 מדריכים - *directories*

לכל קובץ במערכת הקבצים נשמר מידע במבנה הנתונים שנקרא File Control Block. 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.

איך מערכת ההפעלה ניגשת לקובץ?

- משתמש ניגש לקובץ דרך המדריך ("כתובת" הקובץ).
- מ"ה שולפת את ה־FCB הרלוונטי.
- ההרשאות נבדקות מול ה־FCB.
- אם ההרשאות מרשות, ה־FCB מספק מידע לגבי מיקום הבלוקים, אחרת תיזרק טעות.

[4] זה המצב היום, אולם ברגעים אלו ממש מפתחים מערכות הפעלה אחרות שיהיה אפשר לסמוך בהן על המידע בזיכרון הראשי.

ניתן לבדוק הרשאות עבור כל גישה, אבל זה מסרבל כי צריך להעביר כל הזמן את השליטה ממ"ה ל-CPU. לכן בדיקת ההרשאות נעשית רק בזמן הגישה הראשונה לקובץ, והמידע נשמר במקום מסויים אליו נוכל לגשת בקלות בפעם הקודמת.

נסיון שני - 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.

הדירקטורי - קובץ מיוחד המכיל רשימת זוגות מהצורה $(inode, file-name)$. בפתיחת קובץ - מעבר על קבצי הדירקטורי וה-$inodes$ לסירוגין. בכל שלב עוברים על קבצי directory ו-inodes לסירוגין, בכל שלב בודקים הרשאות, וכל מדריך בדרך צריך להיות executable. דוגמא לאיתור קובץ ביוניקס - שקופית 16. בכל צעד נבדקות ההרשאות, עד שהקובץ נסגר ההרשאות נשמרות.

הצבעה ל-inode בזיכרון לכל תהליך יש טבלת file descriptors. כל fd מצביע לכניסה בטבלה גלובלית המכילה file pointers, כל אחד כזה מצביע לכניסה בטבלת in-core inodes.

כל פעולת open יוצרת file pointer פרטי. fds שונים יכולים להצביע לאותו ה-file pointer - ראינו בתרגול.

$Mount$ - 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 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.

5.4 עקביות בעת התאוששות מנפילות

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.

לכן מפתחים משפרים כל הזמן את מערכת ההפעלה, בדרך שגם תייעל את העבודה, וגם תשמור על עקביות בעת התאוששות מנפילות. זהו תחום שמשתנה עם השנים, שכן המהירויות משתנות, גדלי הקבצים גדלים, וכו'.

הכי 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 (למרות שכשאנו עושים את זה, לא תמיד המידע נשמר על הדיסק באותה שניה אלא בזמן קרוב בשל אופטימיזציה של מערכת ההפעלה, יש דרכים לעשות forcing - להכריח את מ"ה לשמור המידע ברגע ספציפי).

לעיתים נכפה $write-through$ - עדכון meta-data ייכתב מיד לדיסק. זה מאט את העבודה.

אם יש נפילת חשמל בעת כתיבה לדיסק, הקובץ נהרס, ולכן לא כדאי במקרים האלו לכתוב כלל. כדי להקטין את סכנת השיבוש במקרה זה, לא כותבים מידע קריטי על המקום הקודם שלו - כדי שאם הכתיבה תהרס נוכל עדיין להגיע לעותק הקודם. אחרי שהמידע החדש הגיע לדיסק אפשר למחוק את הקודם. יש לזה יותר overhead שכן יש כאן יותר פעולות, לכן עושים את זה רק למידע קריטי.

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). במערכות קריטיות זוהי דרך העבודה.

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.

ישנן תוכניות ייעודיות לבדיקת עקביות מערכת הקבצים. ביוניקס - fsck, רצה לבד, ללא גישות מקביליות, קוראת את כל ה-meta-data, ובודקת עקביות מידע לגבי כל בלוק וכל קובץ. כיצד? (שקופית 24) שני מונים לכל בלוק - in-use, free - בודקים ששתי הרשימות חושבות שאותם בלוקים ריקים, או שלא נעשה נקיון פעמיים.

במצב עקבי, כל בלוק מכיל 1 באחד המונים ו־0 בשני. מקרה בעייתי - שני מצביעים קיימים לקובץ, אבל כתוב שהוא $free$ (דוגמא d) - במקרה זה מערכת ההפעלה, ואז צריך להחליט מה לעשות.

כיצד מבצעים הבדיקה?

בונים טבלה עם מונה לכל קובץ, מבקרים בכל מערכת הקבצים רקורסיבית מה־root, ומגדילים reference count בטבלה לכל קובץ. בסיום, משווים המונים בטבלה למונים ב־inodes. אם המונים שווים המערכת עקבית, אחרת לא ויש לפתור בעיות.

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.

5.5 השפעת שינויי הטכנולוגיה

ככל שפערי המהירויות בין דיסק לזיכרון גדלים, המחיר היחסי של גישה לדיסק עולה.

כל השינויים האלו גורמים למפתחים לשקול בשנית כיצד לגרום למערכת לפעול ביעילות רבה יותר.

ככל שהזכרונות גדלים, ניתן לספק את רב הקריאות מה־cache, ולכן עיקר הפניות לדיסק הם בעיקר לכתיבה (או כתיבת הלוג). כאשר חל ההבדל הזה, נרצה שיטות לניהול מערך הקבצים בו הכתיבה היא יעילה יותר.

ה־I/O יותר יעיל ביחידות גדולות ורציפות, בעוד workload של מערכות קבצים מורכב לרב מפעולות כתיבה קטנות ומופוזרות. למשל, בלוק חדש של קובץ, בלוק של ה־inode שמצביע עליו, בלוק ב־free list, בלוק של מדריך שמצביע ל־inode....

כיצד נגיע לנצילות טובה של הדיסק? כל הדברים האלה גורמים לשינויים בתפיסה.

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 האלה. איך נסתכל בצורה אחרת על מערכת הקבצים?

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).

זהו רעיון משנות התשעים, אז לא היה לו שימוש. מכיוון והטכנולוגיה השתנתה, כעת הרעיון מתאים לטכנולוגיות מסויימות כמו אלו שהזכרנו מעלה. כמויות הכתיבה מפוזרות על כל הדיסק, ואז ההתקן יוכל לשרוד יותר זמן.

מה הבעיות? יש overhead, פחות יעיל בקריאה:

צריך למצוא את ה־inode במקום האחרון בלוג בו הוא נמצא. 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 תוך כדי.

הלוג ציקלי, אסור לראש להשיג את הזנב. תהליך נקיון מקצר את הזנב כדי שיהיה לראש לאן לגדול. עבור כל רשומה בזנב, אם ה־inode לא תואם למפה, או הבלוק לא מופיע ב־inode, הוא ניתן למחיקה (הוא ישן) - קדם זנב.

אחרת כתוב את המידע מחדש בראש הלוג, עדכן מפה או inode רלוונטי, ואז קדם זנב. אם הדיסק מלא, נתקענו, הדיסק מלא. יש לדעת בניהול שאנחנו מתקרבים לדבר כזה כדי "לצעוק" למשתמש. לסיכום - 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.

זוהי צורה איתה מתחילים לעבוד יותר ויותר.

5.6 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, 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.
ראינו שמערכות שונות מספקות סמנטיקות שונות.
מערכת אחת לנושא זה - AFS, אשר פותחה בשנות השמונים, משתמשת ב-session semantics - הקבצים שמורים במערכת אוניברסלית, כאשר משתמש רוצה לגשת הוא לוקח בעלות וכאשר משחרר כל אחד יכול לראות את השינויים. 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.
בוינדואז יש מערכת מקבילה, אך אנו נתמקד ב-NFS.
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. דוגמא - שקופית 43.
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.

הרכיבים של המערכת - שקופית 46.
נשאלת השאלה, האם לשמור עבור כל לקוח מצב בין הגישות? למשל, אילו קבצים פתוחים, מיקום בקובץ, מנעולים וכו'.
אופטימלית, נרצה לשמור תמיד - $stateful$.
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.

איך ממשים גישות? משתמשים בשירות Remote Procedure Call (נראה בשיעור הבא או עוד שניים על הנושא הזה) - הפעלת פונקציה במחשב מרוחק (שרת).

כמעט כל קריאות המערכת נעשות כך, אבל לא קריאות וכתיבות, שכן אין להן חשיבות במחשב המרוחק.

פניות למחשב מרוחק מאפשרות לבצע את הפעולה כמה פעמים מבלי לשנות את המצב - תקלה. ואז, אם כותבים פעם שניה או שלישית את אותו הדבר, המצב נשאר כמות שהוא. כלומר - ביצוע כפול שקול לביצוע יחיד.

NFS מבצע *caching* מאסיבי על מנת לשפר ביצועים. נשמרים אצל הלקוח חלקים מהקובץ שאיתו הוא עובד. עדכונים נאגרים בצד הלקוח ונשלחים לשרת מידי פרק זמן, כלומר כתיבות לקובץ אינן נראות מיד אצל לקוחות אחרים.

לא מובטחת סמנטיקה (בניגוד ל-AFS) - כלומר אם התבצע שינוי לא יודעים זאת. בקיצור נמרץ - טוב לשקיפות, רע לסמנטיקה ועקביות.

יש עוד שיטות - Server Message Block - SMB, המבטיחות סמנטיקה יותר טובה. ממומש במערכות רבות כגון Samba, CIFS, אבל לא נרחיב עליו.

שיתוף קבצים

כיום נפוצות מערכות בהם מחשבים משתפות דיסקים.

- NAS - מחשב יעודי עם שטח אחסון גדול.
- SAN - גישה ישירה (טכנולוגיה מתקדמת), יותר מהיר ויותר יקר מ-NAS, יש הרבה סטארטאפים בארץ העוסקים במערכות שכאלו, אך הנושא הזה עדיין לא פרץ (ולא ברור שיפרוץ).

6 I/O ותקשורת בין תהליכים

6.1 מערכות קלט\פלט - I/O

7/6/2012

אנחנו מכירים הרבה התקני I/O:
מחברת, עכבר, מודם... חלקם מעבירים תו־תו, חלקם מעבירים אחרת.
בעבר היה פרוטוקול לעבודה עם כל התקן בנפרד. היום יש אבסטרקציה שאינה מבדילה בין קבוצות של אמצעי קלט\פלט לבין האמצעי עצמו.
(שקופית 6) היום הכרטיס הגרפי הופך היום לאמצעי חישוב פופולרי - פעולה מטריציונית זול יותר לבצע דרך הכרטיס הגרפי. האבסטרקציה היום היא למעשה כבר ארכאית - כל אמצעי הופך להיות היום ל"מחשב קטן" לכשעצמו. במערכות הפעלה מאוד מפוקסות משתמשים באמצעים אלו לחישובים שונים. הסיבה לכך - ה־CPU עצמו מאוד כללי וצריך לתת מענה להרבה אפליקציות, ועל כן לא יעיל במיוחד עבור פעולות ספציפיות, על כן כדאי לפעולות מסויימות להשתמש באמצעים שהם מיועדים לפעולות ספציפיות.
להתקני I/O יש בדר"כ חלק אלקטרוני וחלק מכני, החלק האלקטרוני נקרא "בקר" או מתאם".
הבקר מקבל מידע באיזושהיא צורת התקשרות שמיד נציין, והיום, בניגוד לעבר (בהם היה כרטיס פיזית לכל בקר), הבקר יודע לעבוד עם מספר דיסקים בו זמנית, ומצד שני למשתמש נעשית אבסטרקציה, ואז אם מחליפים דיסק בדר"כ אנו כמשתמשים לא צריכים להחליף כלום. למשל, בקר דיסק יקבל פקודה "כתוב בלוק מידע", ויתרגם זאת להזזת ראש הדיסק למקום וכתיבה למקום זה. כאמור היום כל בקר יכול לשמש כאמצעי חישוב.
ההתקן מכיל רגיסטרים מיוחדים לתקשורת עם המעבד - control, status, input, output.

שיטות גישה להתקן: ההתקן מחובר לפורט $port$ - כתובת וירטואלית שמהווה את כתובת ההתקן ב־bus. מערכת ההפעלה לא יודעת בדיוק מה מסתתר מאחורי הפורט - אלא רק "באופן כללי". ההתקן עצמו ממופה לזיכרון (MMIO), ויש אפשרות דרך אותו מיפוי להגדיר העברת מידע הלוך־חזור. גישה מה־CPU להתקן היא דרך כתובות במרחב הכתובות (כתובת רגילה עם אפשרות להוסיף פסיקה ש"תעיר" את הת'רד שאמור לטפל בזה).
מה Memory Mapped I/O נותן לנו?

- לא צריך פעולות אסמבלר מיוחדת, מאפשר תכנות בשפת־על. עדיין המעבד צריך "לדעת מה הוא עושה" בגישה להתקן, עדיין מפעילים "משהו מאוד ספציפי", ובהמשך נראה שגם את זה יודעים להכליל.
- שימוש בהגנה שקיימת ב־page table = מגדירים כתובות שגישה אליהן גורמת לפסיקה במידת הצורך.
- חוסך פעולה מיוחדת להעתיק רגיסטרים של בקר כי ניתן לבצע פעולות ישירות עליהם.

6.1.1 סוגי התקנים

- התקני בלוקים - block devices:
  - מעבירים בלוק שלם בכל פעם.
  - addressable - לכל מידע יש כתובת.
  - יש גישה אקראית random access - seek.
  - דוגמא - דיסק.
- character devices:
  - העברת מידע ביחידות של בתים־ סטרים של תווים.
  - אין $non-addressable, random\ access$.
  - למשל עכבר, מקלדת.
- $Network\ Devices$
  - כרטיסי רשת מסוגים שונים - אלחוטית\קווית.
  - מעבירים מידע, אבל לאחר מכן לא בטוח שהמידע יהיה קיים אצלנו (פירוט בהמשך).
  - לא התקני בלוק - לא addressable, אין גישה אקראית.
  - העברת מידע לרוב ביחידות גדולות.
  - היום מבצעים אסטרקציה ושינויים להתקנים אלו.

6.1.2 תמיכה בהתקנים

ראינו שיש מגוון גדול של התקנים עם מאפיינים שונים. התמיכה משלבת סטנדרטים כלליים (משותפים לכל ההתקנים) עם דרישות ייחודיות לכל ההתקנים. היום מתפתחת יותר ויותר חומרה שיודעת לעשות פעולות, מצד שני יש אבסטרקציה ש"מרחיקה" אותנו מהחומרה הזו.

התמיכה משלבת סטנדרטים כלליים (שמתרחבים עם הזמן) עם דרישות ייחודיות לכל ההתקנים. בעבר לא היה סטנדרט בכלל. ככל שהאמצעים יותר שימושיים, יש גופים שכל מטרתם היא לקבוע סטנדרטים. לפעמים היא בחזית המידע, לפעמים יש פיגור, כאשר יש פיגור נוצרת תלות ביצרן.

התמיכה היא שילוב של תוכנה וחומרה. המגמה עם השנים - יותר חומרה, פחות תוכנה.

תפקיד מערכת ההפעלה - ניהול משאבים, וממשקים נוחים לתוכניתן (אבסטרקציות) עבור גישה ל-I/O.

האבסטרקציה היא מכיוון והמחשב עובד יותר חזק - יש כאן מעגל.

מצד אחד היינו רוצים להגיע לניהול מאוד מאוד יעיל, אבל גם לשמור על אבסטרקציה - יש כאן ניגוד אינטרסים: שינוי ממודל אחד למודל אחר עם ניהול מאוד יעיל מצריכה שינויים, דהיינו אבסטרקציה נמוכה.

כלומר, יש ניגוד אינטרסים בין כלליות לבין יעילות. הרווח שולי בהגדלת היעילות ומגדיל מאוד את הבעיה בהחלפה בין מודל למודל.

הגישה הכללית לפתרון: עבודה ברמות.

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.

מודל אפשרי - שק' 14:

הקו הארוך מבדיל בין *user* ל-*kernal mode*. מתחת למכשיר יש שכבה שעושה אסבטרקציה לעבודה עם *device*, מתחת יש שתי שכבות ייחודיות לכל התקן - device drivers ו-interrupt handlers, ולבסוף עבודה ישירה עם חומרה.

נכיר את השכבות שעדיין לא פגשנו:

6.1.3 Device Drivers

התקשורת עם התקנים נעשית ע"י device drivers - לכל התקן יש device driver משלו. הם יכולים להכתב בנפרד מהקרנל, או כחלק ממנו, ובכל אופן הם רצים בקרנל מוד.

רב המסכים הכחולים בווינדואז נוצרים מבעיות ב-Device Drivers. זה קרה (כנראה) כי מי שכתב אותם לא אותם אנשים שפיתחו את מערכת ההפעלה. היום כמות הבעיות קטנות, נראה זאת בשבוע הבא - בעיות ב-Device Drivers כבר לא מפילות לנו את כל המחשב (טעויות קטנטנות היו גורמות לבעיות גדולות).

כאמור לכל סוג התקן יש device driver משלו, והם שונים מהותית זו מזו. ההתקן רץ בקרנל, 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 - פעם בעת פתיחת מחשב כל ה-devices היו נדלקים - היום חלק מהניהול של האמצעים כולל הורדת קצב או ניתוק התקן, כאשר המטרה היא חיסכון בחשמל. זהו עולם מתפתח.,לכן בעתיד ה-devices יצטרכו לצפות מתי הם יצטרכו לעבוד. יש משמעויות גדולות לזה - מחשבים ישרדו יותר זמן, למשל.
- הקצאה\שחרור משאבים אוטומטית - היום לא צריך לעשות בוט כאשר מחברים מכשיר חדש (USB למשל). מצד שני זה מאוד מסוכן - יותר קל להתקיף את המחשב.

בעבר הממשק שמ"ה הציעה היה מצומצם. הם היו מסורבלים וקשים לתכנות, ושונים מאוד זה מזה בשל ההבדלים בין *buses* והתקנים שונים.

היום בעזרת הסטנדרזציה מאוד קל לחבר ולנתק התקנים (בעבר היתה פעולה מסובכת). מצד שני, היום אפשר לדמות "כאילו" אנחנו מריצים דיווייסים - למשל אפשר להריץ מערכות הפעלה שונות מתוך אותו המחשב, ולכיוון הזה העולם הולך היום. כמו כן, גם מערכת ההפעלה מספקת יותר שירותים.

6.1.4 ניהול התקנים

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 (שק' 19).
- "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.

6.1.5 העברת מידע בין התקן ל-CPU

שתי אפשרויות:

- MMIO - 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.

נרחיב קצת על בקרת DMA -

אמצעי החיבור נקרא *bus*, כל *device* יכול להעביר דרכו מידע הלוך וחזור. פעם היו מחברים בחוטים כל *device* למקום שצריך, היום משתמשים בפלאג קטן לחיבור ל-*bus*. היום אפילו משתמשים בו בהעברת מידע מ-*device* ל-*device*. אין התערבות של המעבד! עם bus-אים ישנים יותר, היה צורך בבקר DMA מיוחד. ניתן להשתמש ב-DMA על מנת להעביר כמויות גדולות של מידע - מהתקן לזכרון RAM ולהפך, וכאמור אפילו בין זכרונות.

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.

"כמה מפתחי מערכות הפעלה צריך להחליף נורה?" -"אף אחד, זו בעיית חומרה!" - על כן גם מתכנני החומרה צריכים גם להכיר תוכנה.

6.2 Disk Caching

(שק' 24) כאשר משתמש פונה לבלוק, הוא מצפה למצוא אותו בזיכרון. הבלוק אם כך צריך להגיע לשם כדי שנוכל לעבוד איתו. 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.

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.

אחת הסיבות לעבודה בצורה שכזו - כאשר כותבים ורוצים חלק מסויים של קובץ. הוא קטן מהרבה מבלוק בדיסק. אי אפשר להביא את הקטע הקטן, צריך להביא את כל הבלוק. את כל הבלוק שמים ב־data cache, ומשם מעבירים את הקטע המבוקש למקום הרצוי בזיכרון.

כאשר יש בקשות קריאה, הבקר בודק אם המידע ב־cache, אם לא, כאמור קוראים בלוק 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.

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.

כתיבה של בלוקים נחוצה לעקביות, ללא קשר להחלפתם - כדי להמנע מבעיות בעת נפילת מתח, למשל.

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 "מלוכלים" לדיסק, ומכבה את הביט.
  - אפשר להכריח את המערכת לכתוב מידע מסויים לדיסק:
    * sync - יוזם כתיבה לדיסק וחוזר לפני שהכתיבה מתבצעת (כמו שרוצים להוציא USB - ומופיעה הודעה כאשר ניתן להוציא אותו).

* fsinc - 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.

6.3 תקשורת בין תהליכים (העברת הודעות)

דיברנו על העברת מידע בין תהליכים המבוססת על זיכרון משותף. יש שיטה נוספת - העברת הודעות - Message Passing:
שירות העברת הודעות ניתן ע"י system calls: תהליך אחד קורא ל-send(message), תהליך שני קורא ל-receive(&message). הקריאה חוזרת עם ההודעה שהראשון שלח ב-message. צריך אבסטרקציה של ערוץ התקשורת ביניהם - communication link.

6.3.1 אבסטרקציה להעברת הודעות - ערוץ תקשורת

מערכת ההפעלה מספקת את האבסטרקציה דרכה ניגשים לערוץ התקשורת בין התהליכים. למשל, socket ו-pipe.
קריאת מערכת מיוחד "מקימה את הערוץ". האבסטרקציה מיוצגת ע"י file descriptor, עליו ניתן לבצע פעולות שליחה וקבלה של הודעות.
ה-pipe נותן אבסטרקציה של stream - רצף בתים, לא הודעות. גם ה-TCP socket שולח רצף של בתים בין תהליכים שיכולים להיות במחשבים שונים. ה-UDP Socket לעומת זו שולח "חבילה" של מידע - אבסטרקציה של רצף הודעות\חבילות.
ערוץ תקשורת יכול לאפשר קשר בין שני תהליכים בלבד - TCP, ויש ערוצים שיכולים להיות משותפים למספר תהליכים - כמו UDP.
האם לכולם מותר לשלוח או רק לאחד? תלוי בסוג הערוץ.
יש הגדרות לערוץ התקשורת - קיבולת: כמה הודעות\בתים הערוץ יכול להכיל.
כמו כן יש לשקול את נושא האמינות - TCP אמין, שומר על סדר, UDP לא אמין.
אפשר להעביר בצורה סינכרונית - מניח קריאות חוסמות:

- כשאין מקום בערוץ, send ממתין עד שיהיה מקום ואז מתבצע.
- כשאין הודעה לקרוא, receive מחכה שתהיה ואז קורא.

או א-סינכרונית:

- קריאות חוזרות (לא חוסמות) ואומרת שנכשלו במקום להמתין.
- בשלב מסויים הצד השני יקבל את המידע.

ניתן לעבור בין מספר ערוצים. לא נרצה לחכות עד שערוץ מסויים יתפנה או שתגיע הודעה מבלי לטפל בערוצים האחרים.
איך פותרים את הבעיה?

- polling בעזרת שימוש בקריאות אסינכרוניות - קריאת מערכת select.
- בדר"כ משתמשים ב-thread נפרד לכל ערוץ כדי לא לעכב את המחשב (מקשיב לערוץ בקריאה חוסמת).
- איתות - התהליך ישן ומקבל פסיקה כשמגיע מידע בערוץ מסויים, או כשניתן לכתוב לערוץ מסויים.

מקרי קצה של ערוץ עם קיבולת מוגבלת:

- באפר בגודל אפס בערוץ סינכרוני. במקרה זה send ו-receive חייבים להתבצע בו זמנית - הראשון שקורא לפונקציה שלו ממתין לשני. נקרא "נקודת מפגש".
- באפר בגודל 1 - מתאים ל-RPC: אני שולח פקודה לביצוע במקום מרוחק. רק פקודה אחת יכולה לעבור, ויש מנגנוני איתותים שונים - פסיקות כשמגיע מידע, למשל. רק כשהיא מסתיימת אפשר להעביר פקודה נוספת - מודל client/server. עוד על כך בשבוע הבא.

6.3.2 אבסטרקציה נוספת להעברת הודעות - תיבות דואר - mailboxes

זוהי אבסטרקציה חילופית לתקשורת בין תהליכים- 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.

מערכת זו תוכננה כמערכת מבוזרת, ועל כן התבססה על הודעות, אולם מתאימה גם למערכות בעלות מעבד יחיד.

בעבודה עם mailbox האבסטרקציה לכל הפעולות איתה היא הודעה - קבלה, שליחה, וכו'.

6.3.3 תקשורת בין תהליכים במחשבים שונים

תקשורת בין תהלייכם במחשבים שונים נעשית דרך קבלה ושליחה של הודעות מעל רשת תקשורת, בעזרת מגוון פרוטוקולי תקשורת. הפרוטוקולים הנפוצים ביותר היום הם TCP/IP - הפרוטוקולים של האינטרנט.

IP - Internet Protocol - זהו פרוטוקול ברמת רשת (רמה 3), ממומש בכל הנתבים באינטרנט, ודואג לנתב חבילות למחשב היעד. למחשב יש כתובת IP דרכה אפשר לתקשר איתו.

DNS - שירות שעוזר למצוא כתובת, ממפה שמות מחשבים לכתובת IP. נדבר על כך בהרצאה האחרונה בקורס.

6.3.4 פרוטוקולים בסוויטת TCP/IP

במודל יש 7 רמות. רמה 4 - transfort היא הרמה הראשונה של תקשורת end-to-end, כלומר בין תהליכים בשתי מכונות הקצה, ללא התערבות הנתבים בדרך.

יש בסוויטה שני פרוטוקולים - TCP,UDP.

יתכנו ערוצי תקשורת רבים המשרתים תהליכים באותו מחשב (עבור אותה כתובת IP), כאשר לכל ערוץ יש פורט המאפיין אותו באופן חד-ערכי.

ACK - קישור של acknowledgement, או "אישור קבלה" - הודעה שמה שניסינו לשלוח הגיע.

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

אין ב-TCP בטחון מלא במצב - אם קיבלנו ACK אנו יודעים שהודעה הגיעה, אבל מי ששלח לא יודע זאת. כלומר, יש אי וודאות שיש לדעת להתמודד איתה. עוד על כך בשבוע הבא.

כאשר עובדים עם מחשב מסויים, אפשר לעבוד עם אותו פרוטוקול עם פורטים שונים.

6.4 Protocol Stacks

6.4.1 המודל

21/6/2012

דיברנו "בערך" עד כה על פרוטוקולי תקשורת, והזכרנו את מודל 7 הרמות. למה למען השם היתה הכוונה? נסביר כאן[5].

תחילה, דוגמא כדי להמחיש את הרעיון:

נדמיין שאנו רוצים לשלוח מייל. ברמה הגבוהה ביותר, שליחת מייל היא העתקה של קובץ זמני המהווה את ההודעה מהשולח לתיקית האימייל הנכנס של מקבל.

יתכן ויש לפרק הודעות ארוכות להודעות קצרות יותר מכיוון 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, התוכנית המנהלת את האימייל תצטרף אולי לפרק הודעה ארוכה ל-packets קטנים בצד השולח, ולחבר אותם בחזרה להודעה אחת ארוכה בצד המקבל.

תהליך זה נקרא packetization. מאחר וזה שירות שימושי, נרצה לכתוב תוכנית נפרדת לטפל בזה. כך, תוכנית האימייל לא תשלח את ההודעה לתוכנה במחשב השני, במקום זאת, נעביר את ההודעה לתוכנית ה-packetization

[5] לא היתה מצגת, אז הרשום מטה מתוך הסיכום של פייטלסון.

המקומית. תוכנית זו תפרק את ההודעה לפאקטים, תוסיף header עם מספר הפאקט לכל אחת, ותשלח אותם לתוכנית המקבילה במחשב השני, שם התוכנית תרכיב מחדש את ההודעה ותועבר לתוכנית האימייל. באופן לוגי, ההודעה עוברת ישירות מתוכנת האימייל של השלוח לתוכנית האימייל של המקבל, אולם במציאות ההודעה "עושה עיקוף" דרך שירות הפאקטיזציה.

נציין שני דברים:

ראשית, תוכנית הפאקטיזציה מתייחסת למידע שהועבר לה כישות אחת שצריכה לעבור את התהליך, היא לא מפרשת את המידע. שנית, הפאקטים שנוצרו לא צריכים להיות כולם באותו הגודל: יש גודל מקסימלי, אבל אם יתר המידע קטן מגודל זה, ניתן לשלוח פאקט קטן יותר.

יש לוודא כי ההרכבה מחדש נכונה התיאור מעלה מניח כי כל הפאקטים הגיעו בסדר הנכון. אבל מה קורה אם פאקט מסויים אבד בדרך, או אם אחת מגיעה שלא בסדר? אם כך, ל־packetization layer יש אחריות נוספת - היא צריכה לטפל בבעיות פוטנציאליות שכאלה.

טיפול בהודעות המגיעות בסדר לא נכון פשוט - כל עוד אנחנו יכולים לאחסן את הפאקטים המגיעים, אפשר לסדר אותם בסדר הנכון על מספרם הסדרתי. אם מתברר כי אחת חסרה, אפשר לבקש מהשולח לשלוח מחדש. כאן אנחנו מגיעים לסימון של אישור קבלה. על כל פאקט שמתקבל נשלח אישור קבלה, ועל כן השולח יכול לזהות את הפאקטים עבורם לא הגיע אישור קבלה כחסרים.

יש צורך ב־routing אם אין קישור ישיר הסימון שהפאקטים אולי נאבדו או התקבלו בסדר שונה משקף את המקרה השכיח בו אין קישור ישיר בין שני המחשבים. ההודעה (כלומר, כל הפאקטים שמרכיבים אותה) חייב לעבור (routed) בין מחשב מתווך אחד או כמה.

מכיוון ו־routing הוא שירות שימושי, הוא יטופל בנפרד. ה־packetizer יעביר את הפאקטים לראוטר, שמוסיף כתובת או routing header, וקובע לאן לשלוח אותם. הראוטר ב־node המקבל בודק את הכתובת, ומעביר את הפאקטים הלאה. בסוף הפאקטים מגיעים למחשב היכד. הראוטר של מחשב היעד מזהה שזהו היעד, מעביר את הפאקטים ל־packetizer המקומי עבור עיבוד.

אפשר להחיל תיקון שגיאות לכל תשדורת התסריט מעלה הוא אופטימי במובן שהוא מניל שכל המידע הגיע ללא פגע. למעשה, לעיתים המידע עובר שינויים במהלך התשדורת עקב רעש בקווי התקשורת. למזלנו, לעיתים זה אפשרי לחשוב על אמצעים לזהות מצבים אלו. למשל, אפשר לחשב את הזוגיות של כל פאקט והכדרים שלו, ולהוסיף ביט זוגיות (parity bit) בסוף. אם הזוגיות לא מתאימה בצד המקבל, מורם פלאג שקרתה טעות, והשולח מתבקש לשלוח מחדש את הפאקט. אחרת, נשלח אישור קבלה לשולח להודיע לו שהמידע הגיע ללא פגע.

זוגיות היא דרך פשוטה, אם כי פרימיטיבית, לתפיסת טעויות, ויכולת הזיהוי שלה מוגדבל. על כן מערכת אמיתיות משתמשות בדרכים מתוחכמות יותר, כגון CRC. עם זאת, העקרון זהה.

באופן טבעי, נרצה תוכנית נפרד שתטפל בחישובים של קוד תיקון שגיאות ובשליחות המתאימות.

לבסוף, ביטים יכולים להיות מיוצגים על ידי אמצעי פיסי כלשהוא ברמה התחתונה, יש להעביר את הביטים בצורה כלשהיא. למשל, ביטים יכולים להיות מיוצגים על ידי רמות מתח בכבל, או פולדים של אור. רמה זו היא נקראת הרמה הפיזית, ולא מעניינת אותנו כאן - אנו יותר מעוניינים ברמות המיושמות על ידי מערכת ההפעלה.

כל התהליך הזה מביא ליצירת protocol stacks מורכבים כפי שראינו, נוח לחלק את העבודה של ביצוע תקשורת למספר רמות. כל רמה נבנית על הרמות מתחתיה ונותנת שירות נוסף לכל הרמות מעליה, ולבסוף למשתמשים. המידע מועבר מטה דרך הרמות השונות, משיג 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. למשל, רמת ה־packetization מוסיפה הדר שכולל את מספר הפאקט ברצף.

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.

קבוע לקבוע סטנדרטים 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 (לאב לא בשימוש).
- Transport - תעבורה: פקטיזציה ואמינות. אם node נכשל בדרך, הבדיקות ימצאו זאת וישלחו מחדש את הפאקט. רמות גבוהות יותר יכולות להניל כי הקשר אמין.
- Network - רשת: ניתוב - העברת מידע ברשת מקצה לקצב. 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'.

6.4.2 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 ב...

6.4.3 איתור ותיקון שגיאות

ביט בדיקת זוגיות ביט 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 בייצוגו בבינארית יש ספרה אחת שאינה אפס. כל ביט בדיקת זוגיות בודקת 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.

6.4.4 Buffering and flow control

6.4.5 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 להתאודד 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 החלקון. 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.

6.4.6 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־host 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.

כתובות 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. עחלק 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 ראוטים - 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.

7 מערכות הפעלה מבוזרות ווירטואליות

14/6/2012

זהו מימד נוסף המשלים הרבה מן הנושאים שראינו בקורס, הנושא הזה הופך לחלק מהותי בהגדרה של "מהי מערכת הפעלה" בעולם המתפתח שאנו חיים בו.
אפשר להסתכל על רשת כמחשבים מחוברים ביחד - את נושא התקשורת נראה בהרצאה הבאה. היום נראה את האספקט של וירטואליזציה - היום הישראלים מאוד חזקים בתחום זה, על כן חשוב להכיר אותו.

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, כל אחד הוא מחשב נפרד עם מערכת הפעלה משלו (יש נושא של מערכת הפעלה הרצה במולטי־פלטפורם, אך לא ניגע בנושא במסגרת הקורס).
- יש דרך מסויימת של שיתוף פעולה ביניהם - אנו נניח כי מדובר ברשת תקשורת, למרות שיש מודלים אחרים.
- "מצבים" משותפים (בניגוד למה שנראה בשבוע הבא) - למשל תוכן קובץ תור למדפסת משותפת...

עבור כל מכונה, משאב שנמצא באותה מכונה הוא לוקאלי - local, משאב שנמצא מכונה אחרת הוא רחוק - remote.
יש הרבה סוגים של מערכות מבוזרות - אנו נראה שלושה סוגים.

1. מערכת ב־data center - מאוד נפוץ היום בארה"ב. קירור בעזרת מים (ליד הנחלים הקרים טבעית). אוסף של מחשבים שנותנים שירות. כל אחד עובד בנפרד או כולם בשיתוף.
2. מערכת שרת־לקוח - מחשב מרכזי והרבה מחשבים שעובדים איתו.
3. מערכת חובקת עולם - אינטרנט, למשל.

ניגע היום בנושאים אשר בחיתוך בין המודלים השונים.

7.1.1 בשביל מה זה טוב?

- שיתוף משאבים - יכולת להגיע לאינפורמציה שנמצאת מאוד רחוק מאיתנו (מאוד לא יעיל לנייד אינפורמציה).
- cost effectiveness של כוח החישוב - משתלם כלכלית, הרבה יותר נוח שיש הרבה מחשבים ומוסיפים עוד כשצריך. בסדרי גודל, לתכנן מחשב אחד עם הרבה מאוד קורים שיוכל להגיע להרבה מידע בצורה מאוזנת זה משמעותית יותר יקר מאוסף מחשבים שמחברים ביניהם. cloud computing - תומך בחישובים כאלה באינטרנט - חלק מחישוב נמצא באיזור מרוחק ברשת שאפשר לגשת אליו, חלק מהמקום מוקצב להחזקת מידע בלבד, איזור אחר לחישובים.
- תמיכה בגידול הדרגתי - incremental growth.
- ניהול עצמאי (אוטונומיה ניהולית) - כל אחד מהחשבים בצורה מסויימת עצמאי, והם משתפים פעולה ביניהם לטוב ולרע.
- שרידות וזמינות: ניצול נפילות בלתי־תלויות - בגלל תקלה אחת לא נופל הכל, אבל תקלות יכולות להתרחש בכל מקום.

לסלי למפורט הגדיר מערכת מבוזרת באופן הבא:
היא מערכת בה נפילת מחשב שלא ידעת על קיומו מונעת ממך לבצע את עבודתך.
על מנת להגביר את האמינות של מערכת מבוזרת, צריך יתירות ($redundancy$) - המידע קיים ביותר ממקום אחד, ויש כמה דרכים לביצוע אותו החישוב.

7.1.2 נושאים לטפל בהם

- נפילות בלתי תלויות (independent failure).
- תקשורת לא אמינה.
- תקשורת יקרה - הרבה יותר מהר להעביר מידע בין פרוסס לפרוסס באותו המחשב, לבין פרוססים הרצים במחשבים שונים
- עם הבטחות גלובליות - לתת למשתמש ולמפתח את ההרגשה שהם עובד במקום אחד. בשביל כך צריך לוודא שכמה שיותר דברים יהיו אחידים. זה מציג אתגרים לא צפויים. עם השנים הנושא נהיה פשוט יותר, אך העולם הזה עדיין עולם של צרות לא פשוטות.

עבור כל מכונה משאב שנמצא באותה מכונה הוא לוקאלי, משאב שנמצא במכונה אחרת - רחוק.

7.1.3 מערכת הפעלה מבוזרת מול Networked

מערכת הפעלה מבוזרת יוצרת אבסטרקציה של מחשב יחיד מעל אוסף מחשבים - למשתמש זה נראה כמו מחשב אחד.
מערכת הפעלה מסוג זה מתאימה למחשבים הנמצאים בסמיכות גדולה אחד לשני. לא נדון במערכות כאלו.
ב־Networked operating system המשתמשים מודעים לביזור, ויכולים לגשת למשאבים רחוקים בעזרת בקשה מפורשת. אנו נדון בשירותים שונים המאפשרים גישה למשאבים רחוקים.

7.1.4 אבסטרקציות עבור מערכות מבוזרות

1. העברת הודעות בעזרת ערוצי תקשורת - קבלה ושליחה של הודעות בעזרת מגוון פרוטוקולי תקשורת, כפי שראינו.
2. remote login - מחשב אחד מגיע למשאבים של מחשב שני בצורה מרוחקת, למשל etlnet,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.
3. file transfer - ftp תומך ב־get/put וטיול בעץ של מערכת קבצים רחוקה.

בכל השיטות מעלה, בסופו של דבר נוצר קשר בין המחשב לפורט, ודרכו מתבצע הרצת אותה אפליקציה\שירותים\הוראות, למשל העברת קבצים. המטרה היא להגיע לאבסטרקציה כמה שיותר גדולה־ שכן מטרתנו היא שהמשתמש ו\או המפתח יעבדו כמעט באותה צורה עם משאבים לוקאליים ומרוחקים, ואולם כאן האבסטרקציות לא שקופות - המשתמש חייב להתנהג שונה. יש פעולות שצריך לעשות לגבי מחשב מרוחק, ונרצה שהן יהיו כמה שיותר פשוטות ושקופות למשתמש המקומי.

4. ראינו כבר מערכות קבצים מבוזרות - חלק מהקבצים יושבים מקומית, חלקם במיקום מרוחק, ואפילו ניתן לקשר בין שתי המערכות.
5. אבסטרקציה נוספת - RPC - נרחיב עליה:

7.1.5 Remote Procedure Call - RPC

עוד אמצעי לאבסטרקציה - במקום לוגין מפורש וסיסמא מפורשת, עושים אבסטרקציה - RPC - אריזה כללית המאפשרת להריץ פונקציה באופן מרוחק באופן הבא:
תהליך שמתחיל לרוץ במחשב "שלי", שולח בקשה למחשב מרוחק, המחשב "שלי" נרדם כי הוא כאילו מחכה ל־I/O. השלוחה שנשלחה למחשב השני מבצעת את הפעולה ומחזירה את המידע המבוקש, ומבחינת מערכת ההפעלה היא מחכה ל־I/O. במקרה הזה אנו כבר יודעים איך לטפל. במודל זה כל מחשב יכול להיות לקוח ושרת.
התוכניתן שכותב את התכנית, כותב פונקציה שנותנת אפשרות להריץ אותה מרחוק, והקליינט יודע כיצד להפעיל אותה מרחוק - לקליינט נראה כמו קריאה רגילה (פחות או יותר) לפונקציה.
כיצד מבוצע?

- הקריאה מייצרת הודעה המכילה את הפורט של השרת אליו היא צריכה להשלח, ובנוסף את שם הפונקציה והפרמטרים.
- המימוש משתמש בשירות העברת הודעות UDP להקטנת overhead, ואז יש לטפל בבעיות שנוצרות.
- לאחר השליחה, הלקוח חסום (blocked, waiting).

- כשההודעה מגיעה לשרת, בשרת מוגדר תהליך\ת'רד המריץ את הפונקציה ושולח את ערך ההחזרה בהודעה ללקוח..
- הערך חוזר חזרה למחשב הלקוח, לאחר מכן הלקוח ממשיך לרוץ.

זוהי אבסטרקציה מאוד נוחה - "כמו אמבה שמוציאה זרוע שחוזרת אליה".
יש כאן הרבה אתגרים:

- well-known port - איך ניגשים לשרת מסויים? רק למקומות הידועים מראש! איך מוצאים את הפורט? יש פורטים קבועים דרכם אפשר למצוא את הפונקציות הקבועות שמחפשים.
- rendezvous daemon - יש שרת מיוחד המקצה פורטים לשרתים ומודיע עליהם ללקוחות ששואלים. לשרת הזה יש פורט ידוע. לקוחות יכולים לבצע RPC של פונקציית "חפש שרת" אליו כדי לברר כתובת של שרת שיכול לשרת את בקשתם. זוהי דוגמא להגברה - amplification.

מימוש כדי ש-RPC יראה כמו קריאה רגילה לפונקציה, מיישמים stubs - ממלאי מקום של פונקציות..

- השרת קורא ל-server stub, שמבצע blocking receive ונשאר חסום.
- הלקוח קורא לפונקציה שנראית רגילה, אבל משתמש כ-stub. ה-stub שולח הודעה לשרת, כאשר השליחה עצמה דרך הקרנל ומכילה את הפרמטרים בייצוג אחיד. לאחר השליחה, הלקוח נחסם בתוך התהליך שהוא יוזם.
- הקרנל בשרת מקבל את ההודעה מרשת התקשורת דרך תהליכון המחכה על הפורט המסויים הזה כדי לתת לבקשות המגיעות דרכו שירות, ומעביר אותה ל-server stub (מעיר אותו). כאשר הסטאב מקבל בקשה בצד של הסרבר, הוא בודק מהי לפי הפרמטרים, ולפיהם יודע מה להפעיל במחשב השרת. כאשר ההפעלה של הפונקציה מסתיימת, הוא מחזיר תשובה לסטאב אצל הלקוח דרף הקרנל.
- הקרנל של הלקוח מעביר את התשובה ל-client stub ומעיר אותו.
- ה-stub פורק את ההודעה ומחזיר תשובה ללקוח.

ה-RPC לא לגמרי שקוף -

- העברת פרמטרים by reference לא אפשרית - ה-RPC מגביל את סוגי הפרמטרים, $copy-restore$ במקום $reference$ - לא כל ה-$states$ נמצאים במיקום המרוחק, לכן אי אפשר להעביר פוינטרים.
- אסור לפונקציה לגשת למשתנים גלובליים, מאותה הסיבה. אסור שיהיו תופעות לוואי.
- ייתכן ייצוג שונה למספרים במחשבים שונים. יש נטיה להגדיר ב-RPC סטנדרט לפתרון הבעיה הזו.

בעיות אמינות - איבוד הודעות, נפילות שרת\לקוח. RPC היה רוצה להבטיח ביצוע exactly-once, כמו קריאה לפונקציה. אבל, בגלל איבוד הודעות ונפילות שרתים זה לא תמיד אפשרי. מה אפשר לעשות?

- מיישמים טיימר שיתעורר ויבדוק אם הכל עבר. אפשר להחליט לשלוח שוב, אפשר להחליט דברים אחרים.
- יכול להיות שהבקשה בוצעה כבר והתשובה לא הגיע, ויכול להיות שהבקשה לא הגיע, ואי אפשר לדעת מה מהם נעשה. צריך להיות מודעים לכך שהבקשה צריכה להיות כזו שתתבצע פעמיים, אך יראה כאילו בוצעה רק פעם אחד.

במקרה של תקלות

- 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 להבחין 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.

Middleware - עם הזמן הרעיון של RPC שוכלל והוכלל במערכות middleware - הגדרת שכבה בין רשת התקשורת לבין תוכניות המשתמש, העוזרת בפתירת חלק מהבעיות. השכבה נותנת ממשק אחיד לתקשורת בין תהליכים בסביבות הטרוגניות.

7.1.6 שירותי Web

service broker - מתווך בין ה"לקוחות" לשרתים המספקים שירות מוגדר.
לקוח פונה לברוקר אחד או כמה כדי למצוא שרת. הברוקר משדך בין השניים, ואז הם מתקשרים בעזרת פרוטוקול סטנדרטי - *SOAP*. התקשורת בין השרת לברוקר ובין הלקוח לברוקר היא בעזרת WSDL - שפה לתיאור הממשק של ה־web service.
זוהי אבסטרקציה לפעולות ברשת שתופסת ומתרחבת יותר ויותר בימינו.

7.1.7 על אמינות וזמינות

במערכות מבוזרות ללא יתירות, האמינות והזמינות נמוכות יותר מאשר במערכת לא מבוזרת. זוהי תורה שלמה אותה ניתן ללמוד בקורס היעודי "מערכות מבוזרות".
מערכות אמינות fault-tolerant משתמשות ביתירות:

- יתירות בשמירת מידע על דיסקים RAID.
- יתירות ברכיבים פיזיים - כרטיסי רשת, שרתי NAS, וכו'.
- שמירת מספר עותקים של מידע read-only (למשל ספריות) בשרתים שונים - automount מוצא אותם.
- לעיתים יש גם שכפול בכמה שרתים של המידע שנכתב, כדי שהמידע יהיה זמין גם התקשורת הולכת לאיבוד.

7.2 וירטואליזציה

נושא של השנים האחרונות, למרות שהאלמנטים היו גם לפני הרבה שנים.
הרעיון - במחשב אחד מדמה מחשב אחר, מ"ה אחרת או אפליקציה אחרת, תלוי ברמת הוירטואליזציה.

7.2.1 מהי מכונה וירטואלית?

עוד רמת הפשטה - אבסטרקציה. תוכנה רצה מעל חומרה, יוצרת אשליה של חומרה אחרת. הממשק זהה לזה של חומרה הפיזית.
נותנת אפשרות "לחלק" את החומרה למספר מכונות.
הנושא הזה לוקח את כל מה שעשינו בקורס עד כה ו"לשחק בכאילו".

7.2.2 מדוע משתמשים במערכות וירטואליות?

- ניצול משאבים
  - קונסילידציה - 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.

IBM מימשו וירטואליזציה מלאה כבר ב־1967. בשנות התשעים הציעה סימולציה של PC מעל המכונות שלה. הנושא זה לעדנה מחודשת החל מ־~ 2005.

7.2.3 מכונה וירטואלית מול מכונה רגילה

במכונה רגילה, התהליכים עובדים מול הקרנל שעובד על החומרה. במכונה וירטואלית, מעל החומרה יש שכבת bare-metal VMM, מעלה מספר מכונות וירטואליות, לכל מכונה קרנל משלה, והתהליכים עובדים מול הקרנל המתאים.

בעבר לא היה מספיק זיכרון כדי להריץ אפילו מ"ה אחת. היום, כשיש הרבה מאוד זיכרון ויכולות חישוב מרובות, אפשר וכדאי לעיתים להשתמש במכונה וירטואלית.

$VMware$ בנתה סביבה אחת שאומרת - במחשב אחד אפשר לארח סביבות נוספות, ולהריץ את הסביבה שלי (שקופית 34). יש להשקיע הרבה מאמץ בפיתוח הזה כדי לוודא שאין בעיות בין מ"ה אחת לשניה.

אפשרות אחרת - מארחים מוניטור, ומעליו מריצים.

$Hypervisor/VMM$ 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 רגיליסף 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).

כל העולם הזה נמצא כרגע בפיתוח, על כן אין עדיין צורה מסויימת מהבולטת ביעילותה. המחשה של סוגי hypervisor - שקו' 36.

7.2.4 מה צריך לעשות $hypervisor$?

- לספק אבסטרקציה של חומרה - אין תוכנות ייעודיות למערכות וירטואליות.
- לנהל משאבים - לחלקם בין המכונות הוירטואליות:
  - מעבד
  - זיכרון
  - התקני I/O

נשמע מוכר?

מ"ה היא לתהליכים מה ש־hypervisor הוא למערכות הפעלה.

7.2.5 דימוי חומרה - אתגרים

- זיכרון - תרגום כתובות מהוירטואלי, לפיזי של האורח, לפיזי של המארח. היום בחומרה יש כבר תמיכה, אבל זה מעבר למה שנכסה כאן.
- דימוי CPU.
  - חל מהפקודות של מערכת ההפעלה ה"אורחת" צריכות לרוץ בקרנל מוד. אבל ה־hypervisor יאבד שליטה אם ייתן לאורח להריץ מש שירצה בקרנל מוד. למשל, חסימות פסיקות תחסום אותן בכל ה־VMs. על כן, אותן פקודות שצריכות לרוץ רק בקרנל מוד יש לבצע ביוזר מוד - איך למנוע בעיות?
- I/O:
  - למשל, כרטיס רשת משותף לכמה VMs.
  - אין לבעיה זו עדיין טיפול מלא, יש מגוון פתרונות בהם לא נדון.

7.2.6 תרגום כתובות זיכרון

איך טבלת הדפים של ה־host ממפה וירטואלי לפיזי של ה־$guest\ VM$?
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.

7.2.7 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.

לסיכום: כל הבעיות במדעי המחשב אם מוסיפים עוד רמה של אבסטרקציה או הצבעה עקיפה - אבל זה יותר בעיות נוספות.

חלק II
תרגולים

תרגול 1 - אינטרפטים, אקספשנים ו־traps

19/03/2012

הזיכרון הראשי ממוקם על צ'יפים בתוך המחשב (מחוץ ל־CPU).
הזיכרון המשני הוא הדיסק. מידע שנשמר עליו לא נמחק כאשר מכבים את המחשב.
בזיכרון הראשי יש פחות מקום מההארד דיסק. ההארד דיסק יכול לקרוא ולכתוב מידע אל ומהזיכרון הראשי.
זמן הגישה לזיכרון הראשי הרבה יותר מהיר מהגישה להארד דיסק.
תוכנות שמורות על הדיסק עד שהן נטענות לזיכרון, ואז משתמשות בדיסק כמקור וכיעד עבור המידע שהן מעבדות.
רוצים לעבוד כמה שיותר עם זיכרון שקרוב ל־CPU - כך המהירות תהיה יותר גבוהה.

הגדרות - קרנל מול פרוסס

- קרנל - הקוד של מערכת ההפעלה. זהו קוד שהוא חלק ממערכת ההפעלה, ונחשב בבחינתנו קוד שהוא trusted - טוב, ונכתב ע"י "גורמים טובים".
- 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. במערכת שיש בה מספר יוזרים זה עוד יותר קריטי - אולי התוכנה מנסה לעשות משהו שיפגע ביוזר אחר.

מהסיבות האלו המעבד נותן הרשאות שונות לקוד שרץ מתוך process ומתוך kernal. process יכול לגשת למקום מאוד מוגבל בזיכרון, ולהפעיל פקודות מוגבלות בלבד מתוך סט הפקודות הקיימות. ההרשאות האלו באות לידי ביטוי בשני מודים של ה־CPU - קרנל מוד ויוזר מוד, בהתאם למה שרץ עליו באותו הזמן.

הגדרה 7.2 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 (למשל system call), 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.

הרבה דברים שירצה לבצע process "אסור" לו - כאן נכנסת מערכת ההפעלה. process יבקש בקשה מה־kernal. בקשה זו נקראת system call. על כך נלמד עוד בהמשך.

הגדרה 7.3 system call היא בקשה של פרוסס פעיל מהקרנל לשירות המבוצע ע"י הקרנל.

הגדרה 7.4 Input/Output (I/O) - כל תוכנה, פעולה או device שמעביר מידע מ או אל ציוד היקפי (כגון מקלדת, עכבר...

אינטרפטים

הרבה מהעבודה של מערכת ההפעלה היא תגובה לאירועים. כיצד היא מזהה אותם?
למשל, הזזה של העכבר - כאשר משתמש מזיז את העכבר, מערכת ההפעלה צריכה לזהות שהתרחשה הזזה, ולעדכן את המסך בהתאם. היינו חושבים לעשות לולאה שתבדוק את זה, ותעדכן בהתאם. לפתרון הזה יש שני חסרונות:

- זהו פתרון מאוד בזבזני במונחים של כח עיבוד.
- אם יש לנו עוד *devices* במערכת, אולי לא נספיק לזהות את ההזזה בזמן (לפני הזזה נוספת) - כלומר נאבד מידע.

הדרך שבה עושים את זה היא interrupts - זהו אות חשמלי שמגיע מ־device למערכת, כאשר הוא מגיע המערכת צריכה להגיב. אז המערכת לא צריכה לבדוק כל הזמן אם האות מגיע, אלא כאשר יגיע אות היא תגיב בהתאם.
אפשר להפריע לפרוסס ע"י אינטרפטים, אקספשנים ו־traps.

הגדרה 7.5 אינטרפט הוא סיגנל למערכת ההפעלה המציין כי אירוע התרחש, ובעקבותיו מתבצע שינוי ברצף ההוראות שמבוצעות ע"י ה־CPU. אינטרפטים הם אירועים שאינם חלק מהריצה הרגילה של התוכנה.

יש מספר סוגים של אינטרפטים:

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

עוד דוגמאות לאינטרפטים:

- סיום העתקה של מידע.
- לחיצה על המקלדת.

כיצד מטפלים באינטרפטים?

נחשוב על זה כמו על פונקציה - כל אינטרפט צריך לגרום לפונקציה מסויימת לזוז. המבנה של המעבד תומכת בפעולה זו (שילוב של חומרה ותוכנה).
זה נעשה בעזרת מכשיר שנקרא interrupt controller - מזהה את הסוג - מאיזה device הגיעה הבקשה (איזה interrupt number גרם לבקשה). ומעביר אותו למעבד.
המנגנון הבסיסי:

- תפיסת האינטרפט - האינטרפט עוצר את ההרצה של התוכנה. הסיגנל מגיע למעבד (יכול לקרות בכל רגע בזמן - גם באמצע סייקל[6]. בשביל לשמור על הלוגיקה של המעבד, הוא לא יטפל באינטרפט עד שיסיים את הסייקל הנוכחי. לכן אינטרפטים מטופלים רק בסוף סייקל).
- העברת השליטה - עוברים ל־kernal mode.
- שמירת המצב הנוכחי - לפני שמטפלים בבקשה, שומרים בין היתא את ה־program counter - המיקום של ההוראה הבאה שצריך לבצע לאחר סיום טיפול באינטרפט[7], כדי שנוכל לחזור למצב לאחר הטיפול. כמו כן, מכיוון וטיפול בבקשה עלול לשנות את הרגיסטרים, יש לשמור גם אותם.
- טיפול בבקשה - ה־PSW משנה את המצב שלו לקרנל מוד (זה משהו חומרתי לחלוטין). במעבד יש טבלה בשם interrupt vector, ששומרת את המיקומים המתאימים לפונקציות של כל האינטרפטים. כך בהתאם לקוד האינטרפט שהתקבל, מתבצעת הפונקציה המבוקשת.
- שחזור המצב הקודם - משחזרים את כל הערכים של הרגיסטרים, גם את ה־program counter, כדי שנוכל לחזור למצב הקודם.
- החזרת השליטה לתוכנה שהפריעו לה, ובחזרה ל־user mode.

דוגמא - שקופית 24.
לאינטרפטים קוראים גם APC - asynchronous procedure call - הוא לא סנכרוני עם הסייקל של המעבד, ואינו נראה ע"י התוכנה שהפריעו לה.

אקספשנים

הגדרה 7.6 אקספשנים הם דומים לאינטרפטים, אבל 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.

סוג ראשון של אקספשנים הן 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.
סוג נוסף של אקספשן - 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

[6] עוד על סייקלים - בדיג'י!
[7] עוד על זה גם בדיג'י.

בכלל שם לב לכך. הריגסטרים מצביעים לכתובת של ההוראה שממנה הגיעה האקספשן. אם טופל בהצלחה, התוכנית מאותחלת לכתובת שנשמרה. אם אין כתובת לחזור אליה, התוכנית מתבטלת.

יש הבדל דק בין שני סוגי האקספשנים הללו:

לאחר אקספשן של שגיאה, על המערכת להמשיך להוראה הבאה בתור, שכן אם היא תחזור על ההוראה שגרמה לשגיאה, ניכנס ללופ אינסופי.

במקרה השני, מערכת ההפעלה תגרום להפעלה חוזרת של הפעולה (שלא יכלה להתקיים קודם לטיפול), כך שהמשתמש לא ישים לב שקרה משהו נוסף.

Trap

הגדרה 7.7 trap דומה לאקספשן - היא מתרחשת בעת ריצה רגילה של התוכנית. אולם בניגוד לאקספשן, היא לא תוצאה של טעות.

היא הוראה המבקשת ממערכת הפעלה לעשות משהו, כלומר יוצרת אינטרפט - זהו יישום של 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. 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.

הגדרה 7.8 system call הוא מנגנון בו משתמשת תוכנית על מנת לבקשת שירות ממערכת ההפעלה. היא מספקת את הממשק בין תהליך לבין מערכת ההפעלה עצמה (fopen למשל).

משהו שכבר ראינו בתרגיל - system calls יכולות להכשל מכל סיבה כלשהיא. יש לבדוק תמיד ערך החזרה, ואם נכשלה לבדוק מדוע בעזרת קוד השגיאה

דוגמא מופיעה בשקופית 31.

תרגול 2 - סיגנלים

הגדרה 7.9 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 ואינטרפטים די דומים ברעיון שלהם, אבל הם שונים - סיגנלים הם כאמור הודעות לפרוססים שנגרמות ע"י מערכת ההפעלה, ומטופלות ע"י הפרוסס. אינטרפטים נוצרים ע"י החומרה ומטופלים על ידי מערכת ההפעלה.

גם סיגנלים וגם אינטרפטים הם אסינכרונים. ביוניקס ולינוקס יש רשימה סגורה של כל הסיגנלים האפשריים (שמות ומספרים).

מה יכול לגרום לסיגנלים? למשל:

- 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 (למשל $SIGINT$, ועוד).

עוד אפשרות לשלוח סיגנל - הרצת הפקודה kill ב־$shell$.

טיפול בסיגנלים

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 - נראה 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). דוגמא במצגת - הפעולה signal $(SIGNIT, catch-int)$ גורמת לשינוי הטבלה - כאשר יהיה סיגנל SIGINT, הפעולה שתתבצע היא catch-int.
"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.

על תזמוני סיגנלים

כיוון שהסיגנלים יכולים לקרות בכל רגע נתון (אסינכרוניים), הם יכולים לקרות בזמן שסיגנל 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 לרוץ עד שחרור החסימה. נרצה לחסום כמה שפחות קוד.
איך מבצעים חסימת ושחרור חסימת סיגנלים? sigprocmask (דוגמא במצגת) - לכל פרוסס יש דיפולט למיקום המאסקים, אפשר לשנות אותם בעזרת הפונקציה הזו (הוספת או הורדה של פקודות למאסק):
SIG_SETMASK - רשימת הסיגנלים החסומים היא הרשימה שנספק, ללא התייחסות למה שהיה קודם לכן.
SIG_BLOCK - מוסיף לרשימת הסיגנלים החסומים (בהוספה לא משנה אם כבר סיגנל מסויים היה בעבר ברשימה, זה לא יוצר בעיה).
SIG_UNBLOCK - מוריד סיגנלים ספציפיים מרשימת הסיגנלים החסומים.
לפונקציה זו יש לקרוא בתוך כל סיגנלר הנדלר על מנת לחסום ולשחרר סיגנלים.

sigaction

נשים לב כי חסימת הסיגנלים מעלה לא מונעת את כל המקרים בהם יכול להתבצע *race* - למשל, יתכן כי לאחר שנכנסו לסיגנל הנדלר אך לפני שהספקנו לקרוא ל־$sigprocmask()$ נקבל סיגנל נוסף. הדרך להבטיח כי לא יהיו *races* בכלל, היא לתת למערכת לאתכל את המאסקים עבורנו לפני הקריאה לסיגנל הנדלר. זה ניתן לעשות בעזרת הסיסטם קול $sigaction()$, המגדירה את פונקציית הסיגנל הנדלר וגם את הסיגנל מאסק בה ישתמשו כאשר ההנדלר מתבצע.

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

action - סיגנל הנדלר+סיגנל מאסק+פלגס (לא נדבר על הפלגס).
כלומר, מגדיר את הפונקציה שיש להריץ כשיגיע סיגנל מסויים, מגדיר את המאסק - המאסק באקשן לא מבקש לחסום דברים עכשיו, אלא כאשר הסיגנל הנדלר ירוץ. אפשר גם להשתמש בו כדי לראות מהו המאסק הנוכחי. דיפולטיבית הסיגנל *sig* גם יהיה חסום כאשר סיגנל מתרחש. לאחר קריאה לפונקציה זו, הפעולה נשארת עד שפעולה אחרת משנה את הבקשה באופן מפורש.
דוגמא - שקופית 20.

תרגול 3 - *Threads*

ת'רד 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. בשביל שלת'רד יהיה מקום מיוחד, יש לשמור נתונים מסויימים עליו:

- מיקום על הסטאק - פרמטרים לפונקציות, ועוד.

- רגיסטרים של הת'רד - למשל PC הוא חלק מהרגיסטרים של ה־CPU.

שאר המשאבים של הפרוסס משותפים לכל הת'ראדים 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'ראים.
- קבצים פתוחים - אם ת'רד אחד פותח קובץ, אזי הקובץ פתוח עבור כל הת'רדים.
- ועוד...

מימושים

שני סוגים:

- Kernal level thread - מערכת ההפעלה מספקת גם תמיכה בת'רדים שהם פנימיים לכל פרוסס.
  - לכל ת'רד יש thread descriptors שמערכת ההפעלה זוכרת - state וכן stack - כשהת'רד נוצר מערכת ההפעלה זוכרת את המיקום של ה־stack המיוחד שלו, וכן נתונים נוספים שעוזרים לנהל את ה־threads.
  - חסרון: צריך לבקש system call כדי ליצור ת'רד.
  - יתרון: ניהול באגים ע"י הקרנל.
- User level thread - מבחינת מערכת ההפעלה אין בכלל ת'ראדים, אלא רק פרוססים רצים.
  - תמיכה בת'ראדים נעשית בתוך הזיכרון של הפרוסס.
  - בדוגמא במצגת כל ההגדרה של הת'ראד נעשה על ה־heap, אפשר לשמור במקום אחד.
  - הפרוסס עצמו או ספריה של ת'ראדים מייצרת איזורי זיכרון בשביל הסטאקים המיועדים לת'ראד.
  - יתרון: לא צריך לבקש system call, ולכן הפעולות יהיו יותר מהירות (אין צורך לבקש דברים מה־CPU)
  - חסרון: באגים.

מימוש User level thread

מה הספריה צריכה לדעת לעשות כדי לממש?

- לשמור רשימת הת'רדים הפעילים.
- להחליף בין ת'רדים:
  - להפסיק את העבודה של הת'ראד הנוכחי.
  - לשמור את המצב הנוכחי של הת'ראד.
  - לקפוץ לת'ראד אחר.

יכולות אלו ממומשות ע"י פונקציות מיוחדות:

- *sigsetjmp* - שומרת את הנתונים: המיקום הנוכחי בקוד, מצב ה־CPU ו־signal mask.
  - ארגיומנט ראשון - המצב לצורך החזרה: שם ישמר כל המידע הנוכחי.
  - ארגיומנט השני - אפס או לא אפס. אם לא אפס, גם ה־signal mask ישמר לארגיומנט הראשון.
  - המיקום השמור הוא בתוך הפונקציה sigsetjmp - כלומר, כשנרצה לחזור למצב, נחזור לסוף של הפונקציה הזו.
  - ערך החזרה:
    - 0 אם חוזרים ישירות.
    - ערך המוגדר ע"י המשתמש, אם כרגע הגענו לכאן עקב שימוש ב־siglongjmp.

- *siglongjmp* - חוזרת למצב השמור, משחזרת את המצב ואת ה-signal mask.
  - ארגיומנט ראשון - סימניה ששמרנו בעבר באמצעות הפונקציה הקודמת.
  - לוקחת את התוכן של הארגיומנט הראשון ומשחזרת אותם. כאשר שמים את הערך החדש של ה-PC, חוזרים לשם ולסטאק של הת'רד. כאמור הקפיצה תהיה לקוד של ה-sigsetjump.
  - אם המאסק נשמר בפונקציה הקודמת, הוא גם ישוחזר
  - הארגיומנט השני יהיה ערך ההחזרה של sigsetjmp שחוזר אחרי ששיחזרנו את המצב. הערך הזה יעזור לנו להבין, במידה ומגדירים מספר מיוחד, להבין מאיפה קפצנו.

דמו בשקופיות 14-10, אפשר לקמפל ולהריץ על המחשבים בחווה:

- switchThreads - הפונקציה שמנהלת את ההחלפה בין שני הת'ראדים.
- main - מאתחל את שני הת'ראדים (לשים לב, זה לא פשוט!).

הדבר החשוב בהחלפה - הסטאק! לאחר קפיצה, כאשר עושים return, המיקום אליו נחזור מוגדר ע"י הסטאק הנוכחי. כאן רואים את החשיבות של ערך ההחזרה - צריך לדעת איך לחזור. לכן גם נוח שהקפיצה חוזרת ל-sigsetjmp - אפשר להגדיר את ערך ההחזרה.

מה נשמר ב-jbuf?

- PC.
- סטאק פוינטר:
  - כל כתובות ההחזרה של הפונקציות.
  - המשתנים הלוקליים הם על הסטאק, לכן כאשר שומרים את הסטאק פוינטר, בעצם שומרים את הכתובת של המשתנים הלוקליים (ולא התוכן עצמו!). לכן מאוד חשוב לא לשמור על סטאקים של ת'ראדים אחרים! אם זה קורה, לא נוכל לשחזר הנתונים (שכן התוכן עצמו אינו נשמר, אלא רק הכתובת).
- signal mask, במידה ונתבקשנו לשמור אותו.
- שאר הסביבה (מצב ה-CPU).

מה לא נשמר?

- הסטאק עצמו! נשמר רק הפוינטר אליו.
- כל המשתנים שנשמרים בצורה דינאמית, שכן הם נשמרים על מיקום גלובלי.
- ערכים של משתנים גלובליים.
- כל המשאבים הגלובליים האחרים.

אתחול נשים לב שהאתחול כפי שהוא מופיע בדמו תקף רק לגבי המערכות בחווה! מה שמופיע הוא עבור הפונקציה f, קוד דומה יופיע עבור g.

- stack counter - במערכת שלנו - בסטאק מתחיל במיקום מסויים ומתמלא אחורה! לכן, יש באתחול לתת פוינטר לסוף, אבל לא בדיוק לסוף - במערכת שלנו המערכת קודם כל כותבת, ואז מעבירה את הכתובת - לכן לוקחים פוינטר לסטאק, מתקדמים עד הסוף לפי הגודל שלו (בתרגיל יהיה נתון), והולכים אחורה את מספר הביטים שצריך לשמירת כתובת. אולי יש טעות בדמו - וצריך לעשות קאסטינג לכתובת לפני\אחרי.
- PC - הכתובת.

translate address - לא לשחק איתה, להשתמש בה כפי שהיא.
צריך לשים לב בעבודה על הסטאק שלו נחרוג מהמקום שהוגדר, שכן לא נקבל stack overflow - זוהי טעות שמערכת ההפעלה מאתרת, וכאן מערכת ההפעלה לא מעורבת, ולכן לא תיזרק טעות, אלא במקרה של חריגה המידע שנשמור יכול לדרוס מידע אחר.

הקפיצה במיין תפעיל את f. התוכנה לא תסיים לרוץ.

אם נרצה ששני ת'ראדים ישתמשו בפונקציה f, מותר? כן! איך נממש? ההבדל הוא בין הסטאקים.

תרגיל 2

מימוש ספריה שמספקת שירותים של ת'ראדים למשתמש, שיוכל להריץ ת'ראדים כרצונו. אנו לא יוצרים ת'ראדים.
הכי חשוב - הספריה שלנו צריכה להחליט איזה ת'ראד ירוץ בכל רגע נתון, לפי הגדרות שניתנו.

מצבים של ת'ראד

- RUNNING - הת'ראד שרץ עתה, תמיד אחד.
- READY - הת'ראדים שמותר להם לרוץ עכשיו.
- SUSPENDED - לא ניתן להריץ כעת, עד שהמשתמש ישלח פקודת resume.
- SLEEPING - לא יכול לרוץ למשך זמן מסויים.

עדיפויות של ת'ראדים

כאשר המשתמש יוצר את הת'ראד הוא מחליט באיזה עדיפות - priority. ככל המספר יותר גבוה, עדיפותו יותר נמוכה.
העדיפות של הת'ראד יכולה להשתנות בעזרת set_priority.
העדיפויות קובעות ל־scheduler איזה ת'ראד צריך לרוץ בכל זמן נתון.

החוקים של ה־scheduler

- העדיפות של הת'ראד שרץ הוא הגבוהה ביותר מבין הת'ראדים שיכולים לרוץ כעת.
- אם ישנם מספר ת'ראדים בעלי אותה עדיפות, כולם במצב READY, נבחר את הת'ראד שחיכה הכי הרבה זמן במצב READY באופן רציף.

שני החוקים הללו מגדירים בצורה מוחלטת איזה ת'ראד צריך לרוץ בכל רגע נתון.
כל הפונקציות שלנו כתובות כך שתמיד יהיה ת'ראד שנוכל להריץ.
רב הלוגיקה החשובה היא בתוך ה־scheduler עצמו, אבל אולי מתרחשים אירועים חיצוניים:

- ת'ראד שרץ נהיה suspended.
- ת'ראד עובר למצב READY.
- עדיפויות משתנות.

יש דמואים - זה שדיברנו עליו, ודמו נוסף המסביר איך להשתמש בטיימרים (בשביל SLEEPING) - מודיע בעזרת סיגנל שאומר שעבר הזמן שביקשנו שיבדוק. במקרה שלנו מדובר ב־system call. צריך גם לחשוב על העניין של חסימת סיגנלים, ולשים לב שסיגנלים לא יתערבבו זה בזה, וכדומה.

תרגול 4 - Pthreads

היום נדבר על ת'רדים של מערכת ההפעלה. זהו אחד הנושאים החיוניים ביותר בתכנות כיום - כיום התפתחות החומרה היא למולטי־קור, ולמערכת שלא יודעת לעבוד עם ת'רדים לא יודעת לנצל את היכולות החישוביות של מערכות כאלו.
מדוע צריך ת'רדים?

- ביצוע עבודות ב־CPU במקביל ל־I/O.
- priority/real-time scheduling - כך ניתן לקבוע זימון בו משימות חשובות יותר יחליפו או יעצרו משימות בעלות עדיפות נמוכה יותר.
- תגובות תוך כדי עבודה.

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.

באופן כללי, יש לכל ת'רד מעין $main$ משלו - פונקציה שהוא מריץ באותו זמן עם השאר. אנו רוצים שלכל ת'רד תהיה חתימת זיכרון קטנה, כדי שמפתחים לא יצטרכו לדאוג לעניין זה (והמשתמש לא ירגיש בכלל שיוצרים את הת'רד). לכן ב-$POSTFIX$ העלות אכן קטנה.

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.

כאשר מפתחים ב-parallel, צריך לדאוג לכל מיני בעיות, למשל:

- תקשורת.
- תלות במידע.
- סינכרון.

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.

יצירת ת'רדים

בהתחלה, ה-main() מהווה את הת'רד היחיד והדיפולטיבי. את שאר הת'רדים צריך ליצור באופן ישיר - דוגמא ליצירה בשקופית 11.

לזכור **תמיד לבדוק ערך החזרה בעת יצירת ת'רד!**

מתי ת'רד מסתיים?

- הת'רד חוזר מהרוטינה שיצרה אותו.
- הת'רד קורא ל-pthread_exit(status).
- כל התהליך מסתיים.

עוד דבר לשים אליו לב - כאשר אנחנו יוצרים ת'רד **אנחנו לא יודעים מתי הוא ירוץ**, זו החלטה של מערכת ההפעלה.

איך עוקפים את זה?

pthread_join מפסיק את ריצת הת'רד הנוכחי עד שת'רד אחד מסויים מסיים, ורק לאחר מכן ממשיך את הריצה. הת'רד שהסתיים יכול להחזיר ערך - אם צויין אחד כזה ב-pthread_exit(void *status).

אם מסיימים את הריצה של ה-$main$ עם הקוד במצגת 16, הוא לא ימשיך לרוץ עד שכל שאר הת'רדים סיימו לרוץ.

ברב מערכות ההפעלה, ה-main הוא ת'רד מיוחד.

Mutex

למה צריך אותם?

היינו רוצים שקריאת נתונים ועדכונם יהיה תהליך אטומי, דהיינו, ללא הפסקות (כמו עדכון חשבון בנק, למשל).

איך נעשה זאת?

Mutex הוא סוג של משתנה מיוחד המנוהל ע"י מערכת ההפעלה. הוא בעל שני מצבים - נעול ופתוח (עוד עליו בשיעור השבוע או בשבוע הבא).

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 מהערכת 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.
דוגמא - שקופית 25-24.
אחת התקלות הטכניות שיכולות להתרחש - Deadlocks! (דוגמא - מצגת 26). 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.

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.
למשל - כאשר וורד עולה, עולה על המסך הדף הראשון, ולאחר מכן נטענים שאר הדפים.
סינטקס - שקופית 31.
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.
דוגמא - שקופית 35-33.

תרגול 5 - $Concurrency$

30/4/2012

כדי להתמודד עם שיכולות לקרות בעבודה עם מולטי ת'רדס, לא נתמודד עם הבעיות שיכולות להיות לסוג מסויים של מבנה נתונים, אלא נדאג שכל גישה למבנה הנתונים שלנו (כל מבנה נתונים) יהיה mutually exclusive: נפריד לפעולות אטומיות ונדאג שהן יתבצעו בצורה נכונה.
הגדרת הבעיה - the critical section problem. לא מניחים שום דבר על ה-scheduler, ונרצה לממש מכניזם כללי שירשה לת'רד להכנס לקטע קריטי ולצאת ממנו בצורה בטוחה (כלומר, כל עוד הת'רד הזה במקום הזה, אף ת'רד אחר לא יכנס אליו, עד שיצא ממנו).
ראינו בתרגיל 2 שאפשר לעשות זאת ע"י חסימת סיגנלים. כמו כן ראינו mutex. שני הדברים הללו מצריכים תמיכה ממערכת ההפעלה. כאן נראה איך אפשר לעשות זאת עם הקוד של process בלבד, ללא תמיכה ממערכת ההפעלה.
הקריטריונים שנרצה:

1. Mutual exclusion - רק תהליך אחד בתוך הקטע הקריטי.
2. Progress - אין דדלוקים: נוכל להבטיח בכל רגע נתון שמתישהו בעתיד process יקבל את מה שהוא רוצה. זה גם לא מספיק, ולכן צריך גם:
3. Fairness - אין starvation: אף process לא מחכה עד אינסוף, כלומר הוא יקבל את הקטע הקריטי בשלב מסויים. אנחנו לא דורשים כאן שזה יתקבל לפי הסדר (למרות שהיינו רוצים שזה יהיה כך).
4. מנגנון כללי - ללא קשר במספר הפרוססים.

האלגוריתם של פיטרסון

נתונים N פרוססים. 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.

בעיות קלסיות של סינכרון

semaphore הוא אובייקט שדורש תמיכה ממערכת ההפעלה. הוא מאותחל עם מספר משאבים. יש עליו שתי פעולות אפשריות:

P - "אני מבקש להשתמש במשאב שה־semaphore משתמש בו".

V - "סיימתי להשתמש, אני משחרר".

אם $P \neq 0$ אז יופחת אחד ממספר המשאבים עד ש־$P = 0$, אז האובייקט ישן עד שחרור של משאב.

נוח לזכור ע"י P - פחות, V - ועוד.

נשתמש באובייקט זה בכל מיני בעיות קלאסיות:

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.

איזה משאב יש בבעיה הזו? N תאים בבאפר. אבל, זה לא בדיוק מספיק. צריך לדעת כמה תאים מלאים וכמה תאים ריקים. מלאים בשביל קריאה, ריקים בשביל כתיבה. לא מספיק semaphore אחד, כי N פחות מספר התאים הריקים הוא לא מספר התאים המלאים, אולי כעת כותבים לתוך תא ריק, למשל.

לכן, צריך כמה אובייקטים שכאלו:

- 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 - יוסיף ל־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!

בעיית הקוראים והכותבים

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 $P(queue)$ בתחילת הקורא?
אם יש הרבה קוראים שמחכים, ואז כותב מגיע, בלי זה הוא יצטרך לחכות שכל הקוראים יתעוררו (כולם יגשו ל־$P(read)$ 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־$P(read)$, 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.

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.
מה הם יכולים לעשות? למשל:

- לקחת צ'ופסטיק מימין ואז משמאל, אם יש לו שניים אוכל ומשחרר.

זהו פתרון לא טוב - כי אם כולם יקחו את שמאל יחדיו, אף אחד לא יאכל (יכול לקרות במציאות אם לא מנהלים בצורה נכונה את המשאבים).
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.

תרגול 6 - ניהול זיכרון

7/5/2012

נזכור כי כתובות הזיכרון של תוכנות תלויות בתזמון שלהם. לכן אפשר להביט בכתובות בשני אופנים:

- כתובת לוגית - נוצרת על ידי ה־CPU, שם אחר: כתובת וירטואלית.
- כתובת פיזית - הכתובת כפי שהיא נראית על ידי יחידת הזיכרון.

MMU הוא המכשיר בחומרה הממיר את הכתובת הוירטואלית לכתובת הפיזית.
בעבר היה אפשר לקבוע את כתובות הזיכרון בזמן קומפילציה, כך שלא היה צורך ב־MMU.
היום, הכתובות נקבעות בזמן ריצה.
היום טעינה נעשית באופן דינמי בשביל יעול - כלומר, לא טוענים רוטינה אם לא קראו לה. למשל, אם משתמשים בוורד ולא משתמשים בפונקציית הציור שלה, היא כלל לא נטענת. אין צורך בתמיכה מיוחדת ממערכת ההפעלה לצרכים אלו, המימוש הוא דרך הדיזיין של התוכנית.

איך בפועל תוכנות שמות את הכתובת בזיכרון?

הזיכרון הראשי בדר"כ מחולק לשני חלקים:

- מערכת ההפעלה.
- תהליכים של המשתמש.

חלק ממערכות ההפעלה פעלו באופן הבא (מקינטוש בשנות השמונים למשל):
לכל תוכנה יש שני פוינטרים נוספים לאלו שאנחנו מכירים - פוינטר לתחילת הזיכרון שמוקצה לתוכנה, ופוינטר המצבע על האורך של הזיכרון. תוכנות כאלו כמובן צריכות לדעת מראש מהו גודל הזיכרון המוקצה לה.
יתרה מזו, במקרה כזה על מערכת ההפעלה "למלא חורים" - מכיוון וכל תוכנה צריכה בלוק רציף, וכשתוכנות מסיימות לרוץ נקבל "חורים" לאו דווקא רציפים בזיכרון.
ישנו 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.

הגדרה 7.10 external fragmantation - פרמנטציה 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 7.11 internal fragmantation - פרגמנטציה פנימית מתרחשת כאשר יש שטחים לא מנוצלים בתוך המחיצות.

פתרון נאיבי ל"מילוי חורים" - לדחוף את הזיכרון כך שהוא כל הזמן רציף. זה מאוד יקר, אבל אפשר לעשות את זה. ישנם מודלים טובים יותר:

שיטות מיפוי

סגמנטציה

סגמנטציה היא סכמת ניהול זיכרון התומכת באופן בו המשתמש רואה את הזיכרון.
מחלקים הזיכרון לסגמנטים - חלקים שונים: תוכנית ראשית, פונקציה, מתודה, אובייקט...
מסתכלים על הביטים שמייצגים את הכתובת הזיכרון לא כמשהו עוקב, אלא מסתכלים על חלק מהם כמייצגים את מספר הסגמנט, וחלק מייצגים את המיקום בתוך כל סגמנט - כלומר, כתובת לוגית מחולקת לשני חלקים: מספר הסגמנט, והאופסט בתוך הסגמנט.
טבלת הסגמנט ממפה את הכתובת הפיזית לשני חלקים - כל שורה בטבלה מכילה:

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

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.

Paging

זוהי השיטה בה משתמשים היום במערכות הפעלה מודרניות.

הגדרה

- מחלקים את הזיכרון הפיזי לבלוקים בגודל קבוע בשם frames (הגודל הוא חזקה של 2). מחלקים את הזיכרון הלוקי לבלוקים באותו גודל בשם pages - דפים. כך פותרים את הבעיה השניה שהופיעה בסגמנטציה - כל תוכנה מקבלת מספר בלוקים בגודל ספציפי.
- עוקבים אחרי כל הפריימים הפנויים.
- על מנת להריץ תוכנית בגודל n דפים, צריך למצוא n פריימים פנויים, ואז לטעון את התוכנית.
- מקימים טבלת דפים שמתרדמת את הכתובת הלוגית לפיזית.
- יכול להיות פרגמנטציה פנימית - התוכנית בגודל מסויים, אבל לאו דווקא משתמשים בכל המקום הזה.

מדוע זה שימושי? (שקופית 21):
את האופסט (מיקום בבלוק) אפשר לבטא במספר ביטים מסויים כך שלא ניתן כלל להביע מקומות שלא נמצאים בזיכרון, וכך אין צורך בבדיקות חריגה.
כמו כן, מכיוון והגדלים קבועים, לא "מתבזבז" לנו מקום ב"חורים", לכן אפשר לגשת לכל מקום במרחב.
כמו קודם, צריך רגיסטרים שישמרו את המיקום של טבלת הפייג'ים, ואת גודל טבלת הפייג'ים (כדי שלא ניגש למשהו שלא הוקצע). צריך גם ASIDS שיודעים להבדיל בין תוכנה לתוכנה.
מכיוון ויש הרבה בלוקים יחסית קטנים, ככל שמוסיפים תוכנות יש צורך בעוד ועוד טבלאות. על כן חיפשו שיטות שונות לבנות את הטבלאות האלו כך שהחיפוש או המקום המוקצה לטבלאות יהיה יעיל.

שיטה ראשונה - הירכיות במקום טבלה גדולה שתפנה לכל מיקום, משתמשים בטבלה של טבלאות: טבלה ראשית המפנה לטבלאות משנה. טכניקה פשוטה למימוש שיטה זו - טבלה בעלת שתי רמות - דוגמא בשקופית 31-33:
מתחילים ברישא בגודל מסויים, שמפנה לטבלת משנה ראשונה, ושוב מחלקים, ומפנים לטבלת משנה שניה, ולבסוף מגיעים למה שחיפשנו.
מבנה זה הופך את הטבלה להיררכית. השיקול - להקטין את "כמות הגושים".

שיטה שניה - 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: 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.

שיטה שלישית - Inverted Page Table מנסה לשלב את שתי השיטות הקודמות:
במקום למפות לכל תוכנה איפה נמצא הזיכרון, נבנה טבלה בגודל מספר הפייג'ים, ואז רק ממפים מה נמצא בכל מקום פיזית (שקופית 37) - לכל pid פשוט מחפשים איפה נמצא מה שאנחנו מחפשים - כלומר כל שורה בטבלה מכילה את הכתובת הוירטואלית של דף הממוקם במיקום הזה בזיכרון האמיתי. זה חוסך זיכרון, אבל מגדיל את זמן החיפוש - משתמשים בטבלת האש כדי להקטין את מספר החיפושים.

זיכרון וירטואלי

נגענו בו בתרגול הראשון, ונדבר עליו עוד בהרצאה - זוהי הדרך של מערכת ההפעלה להתמודד עם כך שיש הרבה בקשות לזיכרון, בעוד הזיכרון עצמו מוגבל.
ישנו ביט גם בפייג'ינג וגם בסגמנטציה בטבלה שאומר אם מה שאנחנו מחפשים אכן שם, או שכבר נמחק.
זהו התפקיד המרכזי של מערכת ההפעלה - מימוש מתוחכם יותר של virtual memory.
המצב ש"מפחיד" להגיע אליו - thrashing - ההארד-דיסק מתחיל "לטחון". קורה משתי סיבות:

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

הפתרון:

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

לכאורה הבעיה נעלמה בשנות האלפיים (יש הרבה מאוד זיכרון), אבל עם הופעת תחום המובייל הבעיות הללו חזרו להיות רלוונטיות - אין שם מולטיפרוססינג, ולכן פרוסס מועבר כולו כגוש אחד, לא בלוק בלוק.
עוד פתרון - swapping - אפשר להעביר את התהליך באופן זמני מחוץ לזיכרון לאיזור בשם backing store, ולהחזיר אותו להמשך הפעלה. ה־backing store צריך להיות גדול מספיק בכדי להכיל עותקים עבור כל המשתמשים.
roll out, roll in - וריאנט של ה־swapping בו משתמשים עבור אלגוריתמי תזמון מבוססי קדימויות - תהליך בעל קדימות נמוכה מועבר מחוץ לזיכרון בשביל שניתן יהיה לטעון תהליך בעל קדימות גבוהה.
המערכת מנהלת תור של תהליכים המוכנים לרוץ. דוגמא - שקופית 43.

תרגול 7 - מערכת קבצים

הגדרה 7.12 קובץ הוא רצף של ביטים, ללא מבנה מבחינת מערכת ההפעלה. הפעולות היחידות עליו הינן קריאת וכתיבת ביטים. פירוש המידע באחריות האפליקציה המשתמשת בו.

הגדרה 7.13 file description הוא handler לקובץ שמערכת ההפעלה מספקת למשתמש על מנת שיוכל להשתמש בקובץ.

הגדרה 7.14 Metadata של קובץ מכיל את השדות הבאים:

- בעלים: המשתמש שהוא הבעלים של הקובץ.
- הרשאות: מי מורשה לגשת לקובץ.
- זמן שינוי: מתי הקובץ שונה בפעם האחרונה.
- גודל: כמה ביטים של מידע יש בקובץ.
- מיקום: היכן המידע של הקובץ ממוקם על הדיסק.

נדון בישום של מערכת קבצים במערכת Unix קלאסית, כל מערכות הקבצים צריכות לפתור נושאים דומים.

Direct Memory Access

כאשר תהליך צריך בלוק מהדיסק, ה־CPU צריך להעתיק את הבלוק המבוקש מהדיסק לזיכרון הראשי. ההעתקה מבזבזת זמן CPU. אם יכולנו לפטור את ה־CPU מביצוע הפעולה הזו, הוא היה זמין להריץ תהליכים אחרים המוכנים לריצה. זוהי הסיבה שמערכת ההפעלה משתמשת בתכונת ה־DMA של ה־disk controller. גישה לדיסק היא תמיד בבלוקים מלאים!
רצף הפעולות:

- מערכת ההפעלה מעבירה את הפרמטרים ל־disk controller (כתובת על הדיסק, הכתובת בזיכרון הראשי, כמות המידע שיש להעתיק).
- התהליך שרץ מועבר ל־blocked queue ותהליך חדש מה־ready queue נבחר לרוץ.
- הקונטרולר מעביר את המידע המבוקש מהדיסק לזיכרון הראשי בעזרת DMA.
- הקונטרולר שולח אינטרפט ל־CPU, המציין כי פעולת ה־I/O הסתיימה.
- התהליך שמחכה מועבר ל־ready queue.

מבנה הדיסק

הדיסק מחולק ל:

- Boot block - עבור טעינת מערכת ההפעלה (אופציונלי).
- איזור swap (אופציונלי).
- Super block - ניהול מערכת הקבצים.
- i-nodes - file metadata.
- בלוקי מידע - המידע של הקבצים.

סופר בלוק

איזור זה מנהל את ההקצאה של בלוקים ב־file system area. בלוק זה מכיל את:

- הגודל של מערכת הקבצים.
- רשימה של הבלוקים הפנויים במערכת הקבצים.
- רשימה של ה־i-nodes הפנויים במערכת הקבצים.
- ועוד..

בעזרת מידע זה, ניתן להקצות בלוק בדיסק עבור שמירת המידע של הקובץ או metadata של קובץ.

מיפוי של בלוקי קבצים

אין זה יעיל לשמור כל קובץ כ־data block רציף. כיצד נוכל למצוא את הבלוקים אשר יחדיו מהווים את הקובץ? כיצד נוכל למצוא את הבלוק הנכון אם אנו רוצים לגשת לקובץ בהזחה (offset) מסויים? איך נוכל לוודא שאנו לא מבזבזים יותר מידע מקום על management data?
אנו צריכים דרך יעילה לשמירת קבצים בגדלים שונים.
התפלגות טיפוסית של גדלי קבצים מופיעה בשקופית 10 - הרבה קבצים קטנים המשתמשים במעט מקום בדיסק, מספר קבצים בגודל בינוי, ומעט מאוד קבצים גדולים המשתמשים בחלק גדול מהמקום בדיסק.

Unix i-node

ב־Unix, קבצים מיוצגים פנימית על ידי מבנה בשם $i-node$, המכיל אינדקס של בלוקים בדיסק. האינדקס מסודר בצורה היררכית:

- מספר פוינטרים ישירים, המופנים לבלוקים הראשונים של הקובץ (טוב לקבצים קטנים).
- פוינטר לא ישיר יחיד - מצביע לבלוק שלם של בלוקים ישירים נוספים (טוב לקבצים בינוניים).
- פוינטר לא ישיר כפול יחיד - מצביע לבלוק של פוינטרים לא ישירים (טוב לקבצים גדולים).
- פוינטר לא ישיר משולש יחיד - מצביע לבלוק של פוינטרים לא ישירים כלופלים (טוב לקבצים מאוד גדולים).

דוגמאות - שקופיות 4־12, ובתרגיל 4.
כיצד עובדת ההקצאה?
הסופרבלוקים פועלים כזיכרון מטמון של רשימה קצרה של inodes. כאשר תהליך צריך inode חדש, הקרנל יכול להשתמש ברשימה זו על מנת להקצות אחד. כאשר inode מתפנה, המיקום שלו נרשם בסופרבלוק, אבל רק אם יש מקום ברשימה.
אם הרשימה של הבלוקים של inodes פנויים ריקה, הקרנל יחפש על הדיסק ויוסיף inodes פנויים נוספים לרשימה. לשיפור ביצועים, הסופרבלוק מכיל את המספר של ה־inodes הפנויים במערכת הקבצים.
כאשר תהליך כותב מידע לקובץ, הקרנל חייב להקצות בלוקים בדיסק ממערכת הקבצים עבור בלוק ישיר או לא ישיר. סופרבלוק מכיל את מספר הבלוקים בפנויים בדיסק עבור מערכת הקבצים. כאשר הקרנל רוצה להקצות בלוק ממערכת הקבצים, הוא מקצה את הבלוק הפנוי הבא מהרשימה בסופרבלוק.
אחסון מידע בקובץ מצריכה:

- הקצאה של בלוקים בדיסק לקובץ.

- פעולות קריאה וכתיבה.
- אופטימיזציה - המנעות מגישה לדיסק על ידי caching של המידע לתוך הזיכרון.

פעולות על קובץ - כולן נעשות דרך ה־file descriptor:

- open: השגת גישה לקובץ. כיצד מתבצעת הפעולה?
  - מערכת הקבצים קוראת את הספריה הנוכחית ומוצאת את ה־entry של הקובץ המבוקש ברשימת הקבצים המתוחזקת על הדיסק.
  - ה־entry מה־data structure נקרא מהדיסק ומועתק לטבלת הקבצים הפתוחים בקרנל.
  - הרשאות הגישה של המשתמש נבדקות ומשתנה ה־fd של המשתמש משתנה להצביע על ה־entry שהוקצה ברשימת הקבצים הפתוחים.

  למה צריך טבלה של קבצים פתוחים? מעקרון הלוקליות, בנקודות זמן קרובות ניגש לקבצים קרובים, לכן הגיוני להשאיר טבלה כזו. כמו כן, הטבלה משמשת לשמירת קריאות לדיסק.
- close: וויתור על גישה לקובץ.
- read: קריאת מידע מקובץ, בדר"כ מהמיקום הנוכחי. כיצד מתבצעת הפעולה?
  - בארגיומנט מועבר fd הזהה את הקובץ הפתוח על ידי הצבעה לתוך טבלת הקבצים הפתוחים בקרנל.
  - המערכת ניגשת לרשימת הבלוקים המכילים את המידע של הקובץ.
  - מערכת הקבצים קוראת בלוק בדיסק לתוך ה־buffer cache (בלוק שלם = לוקליות במרחב).
  - מספר הביטים בארגיומנט מועתקים לזיכרון של המשתמש לכתובת המצויינת בארגיומנט השני של הפעולה.
- write: כתיבת מידע לקובץ, בדר"כ במיקום הנוכחי. כיצד מתבצעת הפעולה?
  - נניח כי רוצים לכתוב 100 ביטים, המתחילים בביט ה־2000 של הקובץ, וכן כי כל בלוק של הדיסק הוא בגודל 1024 ביטים.
  - אזי, המידע שאנו רוצים לכתוב מתפרש מתחילת הבלוק השני עד ההתחלה של הבלוק השלישי.
  - יש קודם כל לקרוא את הבלוק המלא לתוך ה־buffer cache. לאחר מכן מתבצעת כתיבה של המידע החדש על החלק שעליו צריך לכתוב (כתיבה על המידע הקיים). כלומר כותבים 48 ביטים לבלוק (הבלוק "השני" הוא בלוק מספר שמונה).
  - שאר המידע צריך להכתב על הבלוק השלישי - אזי הבלוק השלישי מוקצה מהרשימה של הבלוקים הפנויים. אין צורך לקרוא את הבלוק החדש מהדיסק. במקום, מקציבים בלוק חדש ב־buffer cache, מסמנים שהוא הבלוק (באטריביוטס הבלוק השלישי מופיע כמספר 2) ומעתיקים את המידע המבוקש לתוכו.
  - לבסוף, הבלוקים ששונו נכתבים בחזרה לדיסק.
- append: הוספת מידע בסוף הקובץ.
- seek: הזזה למיקום מסויים בקובץ.
- rewind: חזרה לתחילת הקובץ.
- set attributes: למשל, שינוי של הרשאות הגישה.

הערה 7.15 buffer cache חשוב לשיפור ביצועים. אבל, הוא יכול לגרום לבעיית אמינות - הוא מעכב את הכתיבה בחזרה לדיסק, ועל כן אם "אין לנו מזל" המידע עלול להאבד.

קריאת המערכת read מספקת כתובת באפר למיקום המידע בזיכרון של המשתמש, אבל לא מציין את ה־offset של הקובץ ממנו המידע צריך להלקח. מערכת ההפעלה מתחזקת את ה־offset הנוכחי לתוך הקובץ, ומעדכנת אותו לאחר כל פעולה. אם יש צורך בגישה רנדומית, התהליך יכול לקבוע את הפוינטר של הקובץ לכל ערך מבוקש על ידי שימוש בקריאת המערכת seek.

ה־file tables הראשיות של מערכת ההפעלה

- טבלת i-node - כל קובץ יכול להופיע לכל היותר פעם אחת בטבלה הזו.
- טבלת הקבצים הפתוחים - מקצים שורה בטבלה זו כל פעם שקובץ נפתח. כל שורה מכילה פוינטר לטבלה ה־inodes ואת המיקום בתוך הקובץ. יכולות להופיע שורות מרובות המצביעות לאותו i-node.
- טבלת ה־file descriptor - נפרדת עבור כל תהליך. כל שורה מצביעה לשורה בטבלת הקבצים הפתוחים. האינדקס של ה־slot הוא ה־fd שהוחזר על ידי הפעולה open.

מדוע שלוש טבלאות?
נניח כי יש קובץ לוג שיותר מתהליך אחד צריך לכתוב אליו. אם לכל תהליך יהיה file pointer משלו, יש סכנה כי תהליך אחד יכתוב על entries ללוג של תהליך אחר. אם ה־file pointer משותף, כל entry חדש ללוג יתווסף בסוף הקובץ. שלוש הטבלאות של מערכת ההפעלה נותנות לנו יכולת לשלוט על השיתוף וההרשאות.
טבלת ה־file descriptor מוסיפה רמה של indirection - כך המשתמש לא יכול "לנחש" קבצים פתוחים וכך לעקוף את ההרשאות.
דוגמא - שקופית 36.

RAID

בעיות עם דיסקים:

- קצב העברת המידע מוגבל על ידי גישה סדרתית.
- אמינות.

פתרון לשתי הבעיות - מערכים נוספים של דיסקים לא יקרים - RAID.
בעבר, RAID (שילוב של דיסקים זולים) היה האלטרנטיבה לדיסקים גדולים ויקרים. היום, משתמשים ב־RAID עבור אמינות גבוהה יותר וקצב העברת מידע גדול יותר. ה־"I" ב־RAID הוא עבור independent במקום inexpensive, וכך היום RAID הוא ראשי תיבות של Redundant Arrays of Independent Disks.

גישות ל־RAID

- RAID 1: mirroring - יש שני עותקים של כל בלוק בדיסקים הרחוקים. שיטה זו מרשה קריאה מהירה (אפשר לגשת לעותק ה־less loaded), אבל מבזבז מקום על הדיסק ומעכב כתיבה.
- RAID 3: parity dsk - בלוקים של מידע מחולקים בין כל הדיסקים בשיטת round robin. לכל סט של בלוקים, מחושב בלוק של parity - בדיקת זוכיות, ומאוחסן בדיסק נפרד. אם הבלוקים המקוריים של המידע נאבדים עקב כישלון של הדיסק, אפשר לבנות מחדש את המידע מהבלוקים הנוספים ובלוק בדיקת הזוגיות.
- RAID 5: distributed parity - בשיטה הקודמת, דיסק בדיקת הזוגיות השתתף בכל תהליך כתיבה, ונהיה ל"צוואר בקבוק". הפתרון - אחסון בלוקי בדיקת הזוגיות בדיסקים שונים.

ניהול I/O ותזמון דיסק

מתייחסים לכונני הדיסק כמערכים חד־מימדים גדולים של בלוקים לוגיים, כאשר בלוק לוגי הוא היחידה הקטנה יביותר של transfer. המערך החד מימדי של בלוקים לוגיים ממופה לסקטורים של הדיסק באופן רציף. סקטור 0 הוא הסקטור הראשון של ה־track ("מעגל" של סקטורים) הראשון בצילינדר החיצוני ביותר. מיפוי מתבצע בסדר לאורך ה־track, ואז על יתר ה־track־אים בצילינדר הזה, ואז ממשיך לצילינדרים הנוספים מהחיצוני ביותר לפנימי.
"איך זה נראה למען השם" אתם שואלים? שקופית 43.
מערכת ההפעלה אחראית לשימוש יעיל בחומרה - קבלת זמן גישה מהיר ו־disk bandwidth (רוחב פס - הביטים המועברים, לחלק בזמן הכולל בין הבקשה הראשונית לשירות וסיום ההעברה האחרונה) גבוה.
לזמן גישה יש שני מרכיבים חשובים:

- seek time - הזמן שלוקח לדיסק להזיז את הראשים לצילינדר המכיל את הסקטור המבוקש.
- rotational latency - הזמן הנוסף בו מחכים לדיסק לסובב את הסקטור לראש הקורא.

המטרה: הקטנה ה־seek time ומקסום רוחב הפס. seektime≈seek distance.
כאשר תהליך צריך I/O מהדיסק, הוא מוציא קריאת מערכת למערכת ההפעלה המכילה את המידע הבא:

- האם הפעולה היא קלט או פלט.
- כתובת הדיסק עבור ההעברה.
- כתובת הזיכרון עבור ההעברה.
- מספר הביטים שיש להעביר.

קיימים מספר אלגוריתמים ליצירת תזמון שירות בקשות disk I/O (דוגמא - שקופית 46-45):

- First Come First Server - FCFS. שיטה זו הוגנת, אבל איטית.
- Shortest-Seek-Time-First (SSTF) - בוחר את הבקשה עם זמן חיפוש seek time מינימלי ממיקום הראש הנוכחי. זוהי צורה של Shortest-Job-First שכבר הכרנו. שיטה זו עלולה לגרום להרעבה של בקשות מסויימות. דוגמא לכל תזוזות הראש - שקופית 48.

השיטה השניה היא המקובלת ויש לה יתרון טבעי על השיטה הראשונה. ישנם אלגוריתמים נוספים המתפקדים טוב יותר עבור מעברות בהן יש עומס רב על הדיסק (מוריד את הסיכוי להרעבה). הביצועים תלויים במספר ובסוג הבקשות. כל הבקשות לשירות מהדיסק יכולות להיות מושפעות משיטת ההקצאה של הקבצים.

תרגול 9 (אין 8) - Unix File System API

על התוכן של הקובץ

28/5/2012

נכיר היום פקודות במערכת הקבצים של Unix. עבור הפקודות שנלמד, שימוש בדפי ה-man: לקרוא את הסקשן השני -

man -S 2 open/read...

- *open* - מקבלת כתובת, פלאגים מסויימים ומוד אופציונלי, ומחזירה מספר שמאפשר לתת הנחיות לעבודה עם הקובץ הזה.
  - פלאגים: קריאה\כתיבה בלבד, O_APPEND - פעולה אטומית, אבל לא ב-NFS(!), יצירה, ועוד...
  - אם יוצרים קובץ, במוד יש לפרט אילו הרשאות ינתנו לקובץ.
- *creat* - יצירת קובץ ודריסת קובץ אם קיים (שווה ערך ל-*open* עם דגלים מסויימים).
- *close* .
- *lseek* - קובעים את נקודת היחס (המקום של הסמן עכשיו), ומגדירים כמה רוצים להתקדם מהנקודה הנוכחית. אפשר גם להתקדם אחורה מסוף הקובץ ע"י מינוס.
  - אם עושים -5 מתחילת הקובץ למשל מקבלים טעות $EINVAL$.
  - אם מתקדמים מעבר להגדרה של הקובץ וכותבים שם, הקובץ גדל אוטומטית, והביטים בין הערך שכתבנו לקובץ המקורי יתמלאו ב-0-אים.
  - מקבלים טעות אם מתקדמים קדימה ממשהו שאינו קובץ (הקלדה למחשב למשל).
- *read* - מגדירים מאיזה קובץ, לאיזה באפר יכנס, וכמה בייטים לקרוא. ערך ההחזרה הוא כמה בייטים נקראו בפועל. אם הגענו לסוף הקובץ יוחזר 0.
  - טעות $EAGAIN$ - ניתן להגדיר בפלאגים גם non-blocking, כלומר עושים read והפונקציה חוזרת ישר (ללא טעות), ואם נקרא שוב ל-read (לפני שפעולת הקריאה התבצעה) יוחזר ערך השגיאה הזה, שאינו שגיאה באמת אלא מיידע שהמידע עדיין לא נקרא. ניתן לבצע לולאה לפעולה הזו עד שיוחזר ערך שאינו השגיאה הזו.
- *write* - מספר דיסקריפטור (לאיזה קובץ לכתוב), מה לכתוב, וגודל הכתיבה. אם ניתן לכתוב את חלק מהבתים ולא את כולם, יכתוב את מה שיכול, ויחזיר כמה בייטים נכתבו.

דוגמא ראשונה - כל האזור בין 10 ל-40 יתמלא באפסים, בין אפס ל-9 יופיע הבאפר הראשון, בין 40 ל-50 הבאפר השני.

דוגמא שניה - תוכנה שמקבלת שני שמות של קבצים, ומעתיקה אחד לשני. לולאת ה-*do* פועלת למעשה עד שהקריאה היא אפס ואיננה טעות.

ניהול קבצים

- $dup2$ - מקבלת שני מספרים שמייצגים דיסקריפטורים של קבצים, גורמת לחדש להצביע לישן, אין שוני בין החדש לישן. אם החדש מתייחס לקובץ פתוח, הוא נסגר קודם לכן. זה per process (שכן הטבלאות בן per process). סגירה לאחד סוגרת גם את השני, וכו'. על פניו זה נראה לא שימושי, אבל משתמשים בזה כל פעם שעושים piping ב־$unix$, בעיקר כשמפעילים תוכנות של אנשים אחרים, למשל $ls|grep\ "my"$.

  - טעויות: אם המספר הראשון אינו דיסקריפטור של קובץ פתוח, המספר השני אינה בטווח המותר לדיסקריפטורים, או אם יותר מידי דיסקריפטורים פעילים.

- $fcntl$ - משנה דיסקריפטורים בשלוש דרכים שונות (דוגמאות בשקופיות):

  - F_DUPED - ינסה להחזיר את המספר המבוקש ב־arg ואם לא אז מחזיר את הכי קרוב אליו, יתייחס לאותו אובייקט כמו fd ויהיה בעל אותו אופסט ואותן הרשאות גישה.
  - F_GETFL - מאפשר קריאה של הפלאגים שהשתמשנו בהם בעת פתיחת הקובץ.
  - F_SETFL - משנה פלאגים מסויימים לערך המופיע ב־arg.
  - בדיקת הפלאגים - בעזרת & (בדוגמא הראשונה).

Links

- $hardlink$ יוצר ערך מקביל לקובץ קיים. לקובץ ב־inode יש counter שסופר כמה ערכים מפנים אליו. אם הערך המקורי נמחק, הקובץ "לא ימות", אפשר לגשת אליו דרך הערך החדש שנוצר. אם מוחקים את הקובץ עצמו, כל הקישורים אליו נמחקים.

  - תפעול זה עוזר מאוד שימושי למשל במערכות גיבוי.
  - יצירת לינק בשל - ln, בסי - $link$. דוגמא בשקופית 29, המספר מציין כמה מצביעים יש לקובץ.

- symlink - $softlink$ - הפניה לערך, יוצר סוג מיוחד של קובץ שתוכנו הוא השם של קובץ היעד. קיומו לא משפיע על הקובץ האמיתי. אם הערך נמחק גם הלינק נמחק - דוגמא בשקופית 31.

- $unlink$ - מוחק את הלינק (הארד או סופט). שקול למחיקת קובץ אם יחיד במקרה של הארדלינק. אם תהליך אחד או יותר מחזיק את הקובץ פתוח כאשר מתקבלת הבקשה למחיקת הלינק האחרון מחיקת הקובץ מעוכבת עד שכל הרפרנס אליהם נסגרו.

- $remove$ עובד על תיקיות.

- $rename$ - שינוי שם.

- פונקציות שנותנות מידע על הקבצים - מחזירות סטראקט עם הרבה מאוד פרטים על הקובץ.

  - $stat$ - מקבל את המידע על הקובץ המצויין בארגיומנט, אין צורך בהרשאות לקובץ, אבל צריך להיות יכולים לחפש ב־pathname המובילה לקובץ.
  - $fstat$ - לא צריך קישור, מספיק הדיסקריפטור של הקובץ.
  - $lstat$ - אותו דבר כמו הראשון, אבל אם שם הקובץ הוא לינק סימבולי, מחזיר מידע על הלינק, בעוד שהראשון יחזיר מידע על הקובץ.

- שלוש פונקציות של הרשאות:

  - $umask$ - מונע לתת הרשאות מסויימות.
  - $chmod$ - משנה את ההרשאות לקבצים מסויימים (במידה ויש הרשאה לעשות זאת).
  - $chown$ - משנה את הבעלים של הקובץ וה־group id.