① 26.02.07
תרגול OS

המתרגלת - שמרית

תרגילים - יש 5 וצריך להגיש את רלם! רצוי בזוגות (לא שלישיות!)

חייבים לבדוק את הניוז - גרוף - כל ההודעות יופיעו שם!

להשתדל להתחיל לעשות את התרגילים מוקדם כי לא רדאי להגיש באיחור!

יש באתר דף שלם של guidelines - חייבים לקרוא אותו!

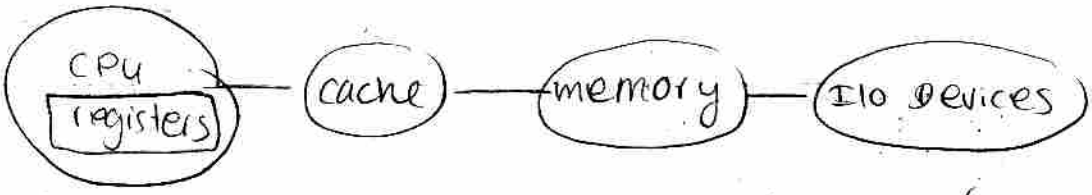
www.cs.huji.ac.il\~os

שקפים בד"כ יופיעו לפני התרגול

דרור בד"כ לא מלמד עם שקפים אבל פרקים מהספר שלו מופיעים באתר, אז רדאי לקרוא אותם קודם.

☺ ☺ ☺ ☺

היררכיה של הזיכרון - יש דיסק קשיח שזה זול ואיטי. ויש את הזיכרון הראשי שהוא דול אבל הוא ברוב למעבד וכל גוב שעובד צריך להיות בזיכרון הראשי. לרוב התהליכים שרדים יש אינטראקציה עם הדיסק וזה יקר לנו כי גישה לדיסק היא איטית מאוד. לק ננסה כמה שיותר שלא תהיה גישה לדיסק אלא ויותר גישה לזיכרון הראשי. כל התרניות יושבות בדיסק וכשהן צריכות לרוץ הן נטענות לזיכרון הראשי.



בין הזיכרון למעבד יש cache שהוא עוד יותר מהיר אבל יותר יקר וקטן והכי קרוב למעבד יושבים האוגרים. כל אוגר מחזיק בעצמו פיסת מידע ובסופו של דבר המעבד עובד ישירות רק עם האוגרים.

Kernel זה כל הקוד של מע' ההפעלה. זו לא תרנית שירצלה לרוץ ולעשות דברים. זאת תרנית אינטראקטיבית. היא יושבת ומחכה לבקשות של המשתמש

ה־kernel נותן בזה בטוח ורשומלפה נמדכי ב־mode kernel הוא
יכול לעשות את כל הפעולות כי הן מטוחות.
system-call מה מקשה ל־kernel על מנ' הפלה. למשל אם
קוראים לפונ' open כי system-call וכשהתכנית עושה
system call המעבד עובר ל־mode kernel ומבצע אותה.

מכל מקום נתון לנו לתלק רק תהליך אחד (מהדוגמה שיש מלפני אותם).
תהליך בזבז מלוק מם פעולות בקטופלט ורשהוא עושה פעולות אלה הוא
עלול לגרום את המעבד אפילו לזוקת הרבה לאן.

אפשר להסביר לתהליך ע' interrupt. למשל אם ישתמשים שלף
אפילו לתדנו את המקלדת אותנו רוצה שהתוכנית "ראה על המסך - אז
הסדרנו לתהליך. תוך מלה אפשר להסביר לתהליך מם ע' exception
interrupt הוא סיגל שאומר למנ' ההפעלה "משהו קרה ואת צריכה
לטפל בו". יש שני סוגים של interrupt - תוכנה (כלה שבא
מבכיב של התוכנה, כמו סיפר או מקלדת וזה אומרן על אלה שבא
מעלות אם) ותוכנה (שמאיים מתוכנה שרלה כ־mode user).
לכל interrupt יש שלhandler שזו הפונקציה שתגיב לתל.
יש שלhandlers של מס' ההפעלה אפילו אפשר לדמיין כזלה מם משלנו.

תכניות בדרך כלל ב־mode-user שלה אומר שהן לא יכולות לגשת
לשום מקום בזכרון שמוקצה ל־kernel או לתוכניות אחרות.
אם תכנית תושה system call אז המעבד עובר ל־mode kernel.

למה צריך interrupts? לתכנית שרלה יש המון פעולות סמיכה-
לא אין התקנים וכו - היינו רוצים לשלב את ההתקנים. אותנו לא רוצים
שכל שהתכנית רלה ימנע מהמקלדת לעבוד. אומרן מה קשה?
הדלה: לא רואה מאן ושאל את כל ההתקנים אם הם צריכים משהו
מלה, לעבוד אפילו מה לכן ישלל כי" וכנל להיות שאז כל ההתקנים רוצים משהו
ותוך מלה אותנו מבזבזים את המעבד שמוטרד לשאול והיא ותכניות.
תוך מלה אותנו רוצים לגרום למעבד כמה מיקס כדי אפ למשל אפשר מאופס

ועד שנשאל אותו ייגמר לו המבוא ב- buffer אז המודע
"אבד.

אז באמת הדבר הזה לא יעלה לכן על אחריות החומרה להודיע רשהיא
דרכה משהו - יזה interrupt של חומרה החומרה מסעונת
שבה בלוח אם ובסוף ה cycle (בסוף פקודה של אסמבלי)
המעבד בודק את זה.

interrupt של תוכנה הוא בד"כ סינכרוני ביחס לתוכנה שיצרה
אותו. כלומר אם נרוץ שוב את התוכנה אז ה interrupt יקרה באותו
זמן. זה נובע מרק שבד"כ ה- interrupt נבע מפקודה מסוימת
בתוכנה.

יש גם interrupt אסינכרוניים של תוכנה שבאים מבחוץ ע"י
סיגנל. למשל אם באמצע ריצת תוכנה לחצנו ctrl-c זה
סיגנל שבא מבחוץ והורג את התוכנה. את ה- interrupt הזה
המתכנת לא ירגל לדרוס (כי אנחנו רוצים שתמיד תהיה אפשרות
לעשות kill) אבל יש interrupts שהמתכנת ירגל לדרוס-
למשל לחלוקה באפס. הכוונה היא לדריסת ה- handlers כמובן.

exception זה סוג של interrupt תוכנה. זה קורה רשהמעבד
מרגיש שהוא לא ירגל לבצע את הפקודה. זה לא תמיד קורה בגלל
משהו רע שהמתכנת עשה ואנחנו אפולו לא בהכרח נדע שקרה
exception. למשל page fault exception קורה כשעי ההפעלה
לא העתיקה את כל התכנית שרצה לזיכרון הראשי. ואז היא הולכת
ומעתיקה את מה שדרוש והכל ממשיך כרגיל כלי שידענו בכלל שקרה משהו.

fault זה כשאחרי ה- exception המעבד חוזר לבצע בדיוק
מאותו מבוא שקרתה בו ה- exception. פעלה זה אומר שהבעיה
טיפלה ואפשר להמשיך מהמבוא שבו הפסקנו. כמו במקרה של page fault

trap קורה ישהתכנית מעבירה את השליטה לעמ' ההפעלה וזה גם מעביר
את המעבד ל- kernel-mode. למשל זה קורה אם יש system-call.

הסוג האחרון של exception זה זה שאותו יודעים שהוא קרה כי התכנית עפה.
abort קורה כשמע' ההפעלה לא יודעת לטפל במה שברה - לאשל
segmentation fault שמע' ההפעלה מודאת או תלובה באפס שהמעבד
מודיע שהוא לא יכול לעשות

הכלה ל- interrupt חומרה. אותנו רודים שהדברים יקרו כמה שיותר
בעילות. מע' ההפעלה דרכה לדעת איזה interrupt קרה ואיזה handler
דרוך להריץ. בשביל זה יש טבלה של ה interrupts והכתובת של ה handlers
שדרוך להריץ. כשמפסיקים תהליך ונותנים ל-handler לחץ חייבים לשמור את
המדב של האוגרים בדי כדי שנוכל לדעת מאיפה דרוך להמשיך (כי ההכלה
של handler יכולה לדרוש את מה שיש באוגרים). לאשל, חייבים
לדעת מה הייתה הפקודה הבאה להכלה.

ה interrupt controller מתווך בין התקני החומרה למעבד. כמו שאמרנו,
יש טבלה שאומרת מה לעשות בכל סוג של interrupt.

מה יודר interrupt חומרה?
- תהליך חידוני שדרוך לבדע
- סיגלים מהתקני קלט ופלט (לחידה על מקש, תזוזה של ערבר)

interrupt חומרה הם אסינכרוניים. הם יכולים לקרות בכל נק' זמן.

השלבים בטיפול ב interrupt חומרה?
- להבין שהיה כזה. (ע"י בדיקת השבב בלוח אם)
- זה עבור שליטה למע' הפעלה (עוברים ל kernel mode)
- לשמור מדב נוכחי (מדב רגיסטרים - PC ועוד)
- לשרת את ה interrupt (לדרוף את ה handler הכלוונטי)
- לשחזר את מדב האוגרים לפני שקרתה ההפרעה
- להחזיר שליטה למשתמש (עוברים חזרה ל user mode).

(3) 12.3.07
OS תרגול

יום הולדת שמח עמית!

היום נדבר על תהליכים.

כשתהליך רץ המידע שלו נמצא ברגיסטרים. למשל ה PC מחזיק את הפוינטר לפקודה הבאה. מרחב הכתובות של תהליך נמצא בזיכרון הראשי RAM. תהליך נקבע גם ע"י הסביבה שלו שמוגדרת ע"י טבלאות במערכת ההפעלה.

לתהליך יש סגמנטים - text - הקוד של התכנית
- initialized data - נתונים גלובליים מאותחלים
- BSS - נתונים גלובליים לא מאותחלים
- heap & stack - זיכרון שמשמש לפונקציות. במחסנית נמצאים משתנים לוקליים שנמצאים בתוך הפונקציה. בערמה מקצים משתנים דינמיים והם לא נעלמים עד שלא משחררים אותם. (הפוינטר לזיכרון בערמה נמצא במחסנית)

לטבלת התהליכים של מ"ה ההפעלה יש פוינטר לסגמנטים האלה של כל תהליך. חוץ מזה יש טבלה של קבצים פתוחים שלה רשום גם דרייברים למיניהם.

איך יוצרים תהליך? הפקודה fork() יוצרת תהליך חדש זהה לחלוטין עד אותו רגע לאבא שלו - לתהליך שקרא לו.
fork() מחזיר את ה PID של התהליך. עבור האבא מוחזר 0 ועבור הבן מוחזר ה PID. טבלת הקבצים וה Inode (עוד טבלה שמזיכה) משותפות לאבא ולבן.

```
res = fork()
if (res = 0)
    // code for parent
else
    // code for child
```

איך תהליך נגמר?

- באופן נורמלי - אם ה. main עושה return או exit.
- באופן לא נורמלי - כשתהליך מקבל סיגנל אם ברתה טעות או משהו.

רשתהליך נגמר באופן נורמלי קורים כמה דברים:

- סוגרים קבצים פתוחים
- משחררים זיכרון
- שומרים סטטוס יציאה
- המלב נכנס למלב זומבי - כלומר עדיין של slot לתהליך בטבלת התהלירים עד שהאבא אסף אותו.

יש macros שמאפשרים להבין איך תהליך מת - בצורה נורמלית או לא.

יש ילדים יתומים - ילדים שהאבא שלהם מת.
ברגע שילד מת הוא זומבי עד שהיא נאסף ע"י האבא. כשאבא מת הזומבים והיתומים מאומצים ע"י תהליך init של המערכת.

סיגנלים מבצעים תקשורת בין תהלירים. הם יכולים להגוע מה kernel או מתהליך אחר. ברגע שיש סיגנל יש signal handler לכל הסיגנלים של handlers דיפולטיביים אבל אפשר לדרוס אותם.

איך שולחים סיגנל?

- ע"י המקשים. למשל ^c.
- ל shell יש פקודה kill.

לפעמים לא כדיך תהליך נפרד. תהליכון (thread) זו יחידת רידה. שחוה בתוך התהליך. בשפול התהליכון משרפלים רק את המחסנית ואת מלב ה רגיסטרים וכל השאר משותף. בתהליכונים מרחב הכתובות נשמר מלך הרבה יותר קל לשתף מידע.
אם ה kernel יודע שיש שני תהליכונים הוא יכול לתזמן אותם, אבל אם לא אז ה user דריך לעשות את זה.

(4) 19.3.07
תרגול 05

היום התרגול עם קובי.

פקודות כמו חיבור ורפל ועשות ישירות על החומרה. רשלרירים שירותים אז מדברים עם מע' ההפעלה. היום נדבר על השירותים שנותנת מע' ההפעלה הרכיבים המרכזיים ב Kernel הם ה- CPU, הזיכרון והתקני קלט ופלט.

נתחיל עם ניהול המעבד. זה אותו רכיב שמאפשר לנו לבדע פעולות. המחשב מנהל את המעבד ע"י תהליך - באופן פשטני כו פשוט אפליקציה שרצה במחשב. טוענים אותם מהדוסק לזיכרון ומפעילים.

תהליך נבבע בעיבר ע"י
- מדב המעבד - הערכים באוגרים
- מרחב הרתובות - התוכן של הזיכרון
- הסביבה - מן קונטקסט שמאפשר להריץ את התהליך.

השמתארים תהליך צריך להפריד בין kernel mode ל- user mode זה עניין של הרשאות ו- security. אנחנו לא רודים שתהליך יכל לעשות כל מה שבא לו.

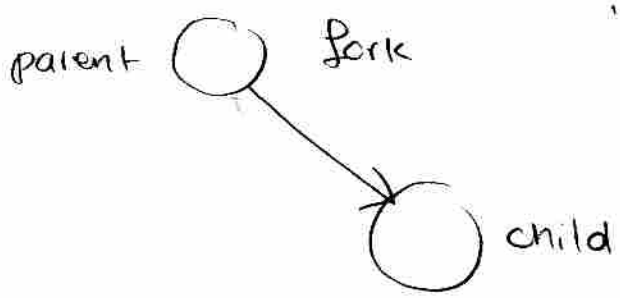
memory

kernel	user
- state	- הקוד - text של התכנית
- memory	
- files - הקבצים הפתוחים	- המשתנים הגלובלייים - data
- accouting	- משמש לזיכרון דינמי - heap
- priority	
- user - נתונים ב המשתמש	- מבנה נתונים בלאסי להרצת פונקציות - stack
- register storage	

CPU

- PC - מצביע לפקודה הבאה
- IR - הפקודה הנורחית
- SP - מצביע לכאש המחסנית
- PSW - אומר מה המצב של המעבד
- רגיסטרים כלליים שמשמשים לכל מיני דברית

רשרודים שירות ממע' ההפעלה עושים system call.
לתהליכים יש מבנה הירכי



השבוראים ל- fork יוצרים בן חדש. יש wrapper שעושה trap וזה מאפשר לעבור ל-kernel mode עליזור בן חדש.
ערך ההחזרה של הבן יהיה 0 וערך ההחזרה גדל האב יהיה ה- PID של הבן (פשוט מספר מזהה שלו). תהליך הבן הוא העתק לוגי זהה של תהליך האב. לפי ערך המוחזר אנחנו מבדולים בין התהליכים:

```
if ((pid = fork()) < 0 )
    // error
if (pid == 0)
    // code for child
else
    // code for parent
```

במצגת יש תיאור של אה שבורה במחסנית בקריאה לפונקציות על- fork. ה scheduler של מע' ההפעלה קובע מי ירוץ אחרי הקריאה ל- fork - האב או הבן.

איך מסיימים תהליך?
תהליך ירוץ להסתיים בדורה נורמלית:
- אם היה return בפונקציות ה. main
- אם קוראים לפונקציה exit ב- C
- אם קוראים ל exit - (זה system call).
במקרה זה יש איזו פרוצדורה שמע' ההפעלה מבצעת.
תהליך ירוץ גם להסתיים בדורה טראגית כמו שאנחנו רבר מרירים, למשל

אם יש segmentation fault.

רשה ידיאה נעשית בדורה נורמלית יש אוניה wrapper. פוובדיית
ה- exit והוא מחזירה לאב את ה- status.
זה מותק את התהליך מהזיכרון, הוא לא ירול יותר לעשית שום
דבר והוא לעולם לא חוזר.
מע' ההפעלה עושה את הדברים הבאים:
- מיסוך סיגנלים - הוא הרי לא ירול לקבל יותר שום דבר
- סגירת קבצים ומשאבים שהתהליך השתמש בהם.
- שחרור זיכרון. כל הזיכרון.
- לשמור את סטטוס הידיאה של התהליך רדי שאפשר יהיה
להעביר את זה לאב.
- הפירת התהליך לזומבי

נניח שהאב רוצה לדעת מה בורה עם הבנים שלו. בשבול זה
יש את הפקודה wait. אם יש ילד זומבי אז wait
מחזיר את ה PID שלו, ואחרת חוזר -1.
ו- wait מרנוס לתוך status את הסטטוס ידיאה

למערכת יש תהליך init עם PID=0. ושאב מסתיים
היתומים והזומבים שלו מאומצים ע"י init.

אם היה לנו רק fork אז הינו ירכלים להריץ כל הזמן רק את אותו
התהליך והיה משעמם מאוד.
בשבול להריץ תכניות חדשות משתמשים ב פקודה
exec(filename, argv, envp)

- אפה שנמצא הקובץ שדריך להריץ - ה- executable
- הארגומנטים שמעבירים לפונקי main התהליך
- משתנה סביבה - הרשאות של המשתמש וכל מיני רעלה.

שלא רמו ב- fork לא חוזרים מ exec שהסתיימה בהצלחה.
כל מה שהיא עושה זה לטעון תהליך חדש והיא דורסת את
המחסנית והסביבה והכל.

מה קורה במע' ההפעלה כשקוראים ל- exec?
- בודקים שהקובץ הוא אכן executable (מקומפל למכונה הזאת),
שלמשתמש יש הרשאה להריץ אותו ושהוא אכן מותאם למערכת
- מעתיק את הארגומנטים למרחב המע'
- לשחרר את הזיכרון הקודם
- להריץ את התהליך.

הפעושים fork לא חייבים לשרפל את סגמנט ה text שהרי
שני התהלירים מריצים בדיוק את אותו הקוד אז שני המלבינים
מלבינים לאותו סגמנט.

האם יש הגבלה על כמות התהלירים שאפשר לידור? אין הגבלה עקרונית
אבל פרקטית יש הגבלה כי המשאבים מוגבלים. כל תהליך לפחות
חייב זיכרון והזיכרון ירכל להיגמר.

ה shell זה מה שמקבל פקודה מה. STDIN ומריץ אותה.
ה shell הוא תהליך האב וכשכותבים פקודה זה יוצר בן.
יש שני סוגי פקודות- פנימיות (פקודות שמתבלעות ישר אלו
תהליך חדש) וחיצוניות (שעושים להן תהליך חדש רדו לבלע).
פקודות חיצוניות אפשר להריץ בשני אופנים- ברקע או לא.

סיגנלים הם הודעות שמגיעות לתהליך רדו לספר לו על מאורעות שקרו.
הם ורכלים להגיע מה kernel או מתהליך אחר. יש system call
שנקראת kill והיא יודעת לשלוח סיגנלים בין תהלירים.

כשתהליך מקבל סיגנל הוא צריך לדעת איך לטפל בו. הברירת מחדל הוא
פשוט לצאת. אבל ישנם סיגנלים שאפשר להתעלם מהם

(6)

למשל אם רוצים לסמור את המחשב צריך להגיד לכל
התרניות להיסמר. ה kernel שולח לכל התרניות שרצות
פקודה להיסמר. זה סיגנל שאי אפשר להתעלם ממנו.
אופציה שלישית הגי שרשתהליך מקבל סיגנל הגו
מבצע איזו פונקציה. מעמירים לו פוינטר לפונקציה
שצריך לבצע - oldfun = signal(signum, newfun)
זה system call שאומר ל kernel אזה קטע קוד
צריך להתבצע.

(7) 26.3.07
ת.מע' הפעלה

יום הולדת שמח לטליף

סיגנלים -

- אלה בעצם הודעות שנשלחות לתהליכים רדי להודיע לו
על דברים שברו. אותו רודים לטפל בסיגנלים בקונטקסט של
התהליך עצמו. signal handler זו הפונקציה שמטפלת בסיגנל.
יש 3 מודלים מרכזיים לטיפול
- התהליך יוצא (זו ברירת המחדל)
- התהליך מחוקלם
- התהליך מבדע פונקציה מסויימת שרשומה בטקסט (כלומר בקוד שלו)
אז ה- PC עובר להלביע לפונקציה הזאת, היא מתבדעת ואז
חוזרים למקום שעדרנו בו. זה הכל באותו קונטקסט כי אנחנו לא
מחליפים תהליך זו כמו סתם קראה לפונקציה.

נניח שקיבלנו סיגנל, קפצנו ל signal handler שלו ואחרי שהתחלנו
לבדע אותו קיבלנו עוד סיגנל. אז נקפוץ ל- signal handler של
הסיגנל החדש. אבל זאת בעיה כי לא סיימנו לטפל בראשון, וזה ירגל
לידיר תוסר תאימות (למשל אם שניהם משתמשים באיזה משתנה
גלובלי). אז בזמן שמבדעים signal handler אותו נרצה למסך
את הסיגנלים האחרים רדי שנבדע את כל מה שלריך כבלוק אחד
ואז נטפל בסיגנלים האחרים. רודים לטפל בסיגנלים בצורה סדרתית.
אז ברניסה ל- handler תוסמים קבוצה של סיגנלים שאנחנו לא
רודים לטפל בהם ובודיאה מהפונקציה משחזרים את המלב, ואז
אפשר לבדע אותם בצורה סדרתית.
יש בעיה: ירגל להיות שלפני שחסמנו את הסיגנלים יגיע סיגנל.
בשבול זה אפשר להשתמש ב sigaction במקום ב signal
וזו פונקציה שמבדעת את החסימה באופן אוטומטי, ואם לא דרך לסנשות
בה reinstall לטיפול בסיגנל.

תהלירון מאפשר להריץ מתוך תהליך בזיכרון קונטקסטים. משמעות זה
לכל תהלירון דרירה להיות מחטנית נפרדת (רדי שהריזה תהיה בלתי
תלויה), לכל תהלירון דרירה להיות רגיסטרים משלו ואישה לכל
המשאבים של התהליך שלו (הוא לא מנהל אותם אבל יש לו גישה).
זה אומר שהתקשורת בין תהלירונים היא נותה יותר. בגלל שלשניהם
יש גישה ל heap הם ירולים לתקשר דרכה ואולו תהלירים
מתקשרים בעזרת sys calls.

יש שני סוגים של תהלירונים.

- kernel level - מע' ההפעלה היא זו שמנהלת את התזמון
שלהם עבורנו.
- user level - למשתמש יש ספריית תהלירים והיא זו שאחראית
על התזמון ועל ה contextswitch. מאקרה זה המע' בכלל
לא יודעת שיש כמה תהלירונים

(8) 28.5.07
ת' OS

תהליכון זו יחידה עדמאית שיכולה לרוץ אבל היא חיה במרחב הרתיבות של תהליך כלשהו. רשיזרים תהליך חדש משרפלים את כל מרחב הרתיבות לך צ פעולה יחסית כבדה וגם שיתוף מידע לא פשוט כ"כ. רש יזרים תהליכון יש דברים שלכל תהליכון דרירים להיות בנפרד (כמו stack, רגיסטרים כמו PC, SP) ואותם יזרים מחדש אבל המון מידע משותף לכולם (heap, משתנים גלובליים) ואז שיתוף המידע פשוט. תהליכון יכל להסתיים רשהוא מסיים את הפיזה שלו, רשתהליכון אחר מסיים אותו או רשהתהליך כולו יוצא ואז כל התהליכונים מסיימים יחד איתו.

לכל תהליכון יש את האטריביוטים הפרטניים שלו:

- Stack
- רגיסטרים
- תכונות של תזמון (עדיפות או policy)
- סיגנלים שמחכים או חסומים

למה דרוך תהליכונים?

- הניהול של התהליכונים קל יותר מניהול תהליכונים.
- שיתוף המידע פשוט יותר כי ה heap יה data משותפים

הספריה pthread יש 3 משפחות של פונקציות

- ניהול תהליכונים - יצירה / עריכה / שינוי
- מנעולים - נעולה ושחרור שמשמשים לתזמון
- משתני מצב - מאפשרים. לפעול ראשר מתקיים תנאי מסוים.

יצירת תהליכון:

ה- main הוא התהליכון הראשון. בתורי אפשר ליצור עוד תהליכונים בעזרת הפונקציה pthread-create שאת הפרמטרים שלה אפשר לבדוק ב- man.

על אלגוריתמים שונים למנעול.
הפונקציות וכל הדוגמאות לשימוש בפונקציה שמופיעה לפני
יש שתי דברים לבחור למנעול. בדרך הכלל מוגדרת עם ה הלוגיקה
ופונקציה לשחרור המנעול.

מיזוג

נתון לנו מערך ממוין של מספרים שלמים. נרצה למזג אותו עם מערך אחר ממוין כך שהמערך החדש יהיה ממוין.

פתרון סדרתי

נעבור על שני המערכים בו-זמנית ונכניס את האיבר הקטן יותר למערך החדש. נמשיך כך עד שנגמרו האיברים באחד המערכים. נכניס את כל האיברים שנותרו במערך השני למערך החדש.

פתרון מקבילי

נחלק את המערך הראשון לחלקים שווים בגודלם. נמצא לכל חלק את האיבר הראשון והאחרון בו. נמצא את האינדקס של האיבר הראשון במערך השני שגדול מהאיבר האחרון בחלק שלנו. נמצא את האינדקס של האיבר האחרון במערך השני שקטן מהאיבר הראשון בחלק שלנו. נמזג את החלקים שמצאנו.

סיום תהליכון

בעיה - התהליכון הראשון לא יודע שהוא אמור לחרוץ. אלא הפונקציה השנייה היא שמפורקת סיאמי לחץ.
הפונקציה השנייה היא הקוראת לפונקציה pthread_exit שמאפשרת את התהליכון שקרא לה.

הפונקציה השלישית היא שתהליך ראשי main סיים לחץ וגם מעוניין להרוג את כל התהליכונים שלו
משתמשים pthread_exit ב- main כדי לגרום את התהליכון main לחכות את כל התהליכונים שהוא יצר כדי להמשיך לחץ.

בדרך תהליכונים:
כשרואים לולאה שיוצרת ל- main סיים עד שכל תהליכון אחר מסיים. אלה עושים ע"י הפונקציה pthread_join.
וגם ה- main אוכה עד שהתהליכון הכלוני מסיים לרוץ.

מיזוג יש הרבה דוגמאות (בהנחה שהמערכים ערוכים ב-FIFO)

קטעים קריטיים - נלמד מכירים את הבעיה המשותפת עם חשבון בנק שמפקידים בו כסף. יש לנו רוצים שהפקדה תהיה פעולה אטומית: בשביל זה קיימים mutex-ים. זה בעצם מנעול שרק תהליכון אחד יכול לקבל. שני תהליכון אחר יבקש את המנעול הישן ויתקע. רק כשהתהליכון שם המנעול מסיים אחרים יכולים לקבל אותו.
אז אם לפני הפעמן נעלנו, ואחרי נפתח הכל יהיה בסדר.
יהיה הבעיה של מנעולים.

(9)

נעולת mutex:

קוראים לפונקציה pthread-mutex_lock ואם אל תהליכון לא מחזיק כבר את המנעול הזה אז הוא בעמת ננעל ומקפלים אותו. אחרת התהליך מחכה עד שהמנעול פנוי.
בשביל לשחרר מנעול משתמשים ב. pthread_mutex_unlock.

משתני מצב -
אנחנו רוצים להיות מסוגלים להכיר עד תנאי מסוים, עד שקרה משהו מסוים. הדרך הפשוט לעשות את זה היא להתעורר כל הזמן, לבדוק את קיום התנאי ולחזור לישון במקרה הצורך. זה נקרא polling וזה מן הסתם לא יעול בכלל. זה בעצם busy-wait ואסור לעשות את זה אל פעם!!!

גם כאן יש אתחול סטטי ואתחול דינאמי. בכל אופן בטל צריך לעשות destroy. תהליכון ירול לגרום לעצמו לגרת לישון ע"י pthread-cond-wait שמקבלת משתנה מצב ומנעול. רשהוא מתעורר מה- wait הוא מקבל את המנעול. לכן צריך לנעול אותו לפני שהולכים לישון. תהליכון אחר בודק את קיום התנאי ואם הוא מתקיים הוא ירול להעיר תהליכון ישן כל שהו ע"י pthread-cond-signal או שהוא מעיר את כולם ע"י pthread_cond broadcast.

במצגת יש דוגמה.

בעיות ופוצות בסנכרון

- יש מערך בגודל קבוע N. יש סמפור mutex שמאותחל ל-1, סמפור full שמאותחל ל-0 וסמפור empty שמאותחל ל-N. האתחול אומר כמה תהליכונים ירולים להחזיק במנעול.
יש יצרן שמתחול להכניס דברים למערך ונתקע רשהוא מלא ויש דרכן שמוציא מהמערך ונתקע רשהוא ריק.

- בעיות בריאה / רתיבה - יש מידע שמשותף לרמה תהליכונים.
קוראים רק קוראים מהמידע ונותבים ירדלים גם לבחוא וגם לרתוב.
רוצים לאפשר לרדלה לעבוד ושהמידע עדיין יהיה עקבי - אז
הרבה קוראים ירדלים לבחוא בו"ז אבל רק כותב אחד ירדל לעבוד
בכל זמן נתון. אבל אורנו לא רוצים להרעיב אף אחד.
במלגת יש פתרונים למברה שרוצים לתת עדיפות לקוראים ולאברה
שרוצים לתת עדיפות לכותבים.

- הפילוסופים הטועדים - יש חמישה פילוסופים סביב שולחן עגול ובאמצע
יש קערת אורז. בין הפילוסופים יש 5 מזלגות בשפיל לארדל
מהדלחת הם צריכים מזלג אחד אבל בשבול להעביר אורז.
מהקערה לדלחת הם צריכים שני מזלגות.
אם כל הפילוסופים פועלים בדיוק באותו אופן

(10) 4.6.07
ת' 05

יש הרבה תהלירים שרצים על מע' הפעלה והם צריכים לתקשר ביניהם.
כבר ראינו איך אפשר לעשות את זה ע"י סיגנלים.
IPC הוא מרניזם שבו תהלירים משתמשים לסנכון את הפעולות
שלהם. יש שתי פעלות - send ; receive /
יש רמה שיטות לתקשר. חלק מהם קודם יוצרים connection ואז
מעבירים מידע. בשיטות אחרות אפשר לשלוח הודעות בלי
לפתוח נתיב תקשורת. לנתיב התקשורת יש מאפיינים פיסיים (ציבון,
חומרה, רשת) ומאפיינים לוגיים (FIFO ; טיפול בטעויות).

תקשורת ישירה -
תהלירים חייבים לציין את הרתובת של התהלך שהם מתקשרים איתו.
במוקרה זה הקישוריות נוצרת אוטומטית בין שני תהלירים.
התקשורת ירולה להיות חד כיוונים (למשל signal) אבל הוא
בד"כ דו כיוונית (socket).

תקשורת עקיפה -
הודעות עוברות תמ"ד דרך תיבת דואר (port). לכל תיבת דואר
יש שם מזהה ותהלירים ירולים לתקשר רק אם יש להם תיבת
דואר משותפת. אז לא רק צוג ירול לדבר, אלא כל מי שמכיר
את התיבה הזאת.

סנכרון -
לפעמים תהליך קורא לפוובציה שלא ירולה להתממש באותו רגע
(wait, lock) ואז התהליך נחסם. זה נקרא סינכרוני.
יש פוובציות שהן non-blocking (trylock) ולהנחשב
לא סינכרוני. גם שרותי תקשורת בין תהלירים ירולים להיות
blocking ו- non blocking. למשל, אם ה- receive הוא
blocking ואין הודעות בתיבה אז התהליך יחכה שתהיה
הודעה. ואם הוא לא non blocking אז תחזור שגיאה.

סיגנלים -

תהליך יכול לעשות סיגנל ולהעביר אותו לתהליך אחר. ה-
signal handler של תהליך הועד וקרא והוא מטפל בזה.
זו תקשורת ישירה ועדיף התקשורת נוצר אוטומטית. התהלירים
חייבים להגיד בדיוק למי הרי שולחים. וזה אסינכרוני!

pipes.

זה זוג של filedescriptors. יש אחד שכותב לתוך pipe
ואחר שקורא ממנו ה- pipe הוא ערוץ התקשורת שלנו. בד"כ
הגדל של ה- pipe הוא 10K. בשפול לתקשר תהליך אחד צריך
את ה fd הכותב והשני צריך את הקורא. מה שקורה בד"כ
הוא שיש תהליך אב שעושה fork ואז האב זוכר את
הכותב והילד זוכר את הבן.

מהבחה אם רוצים תקשורת בין תהלירים שהם לא אב ובן. למקרה
זה יש named pipe. ואז תהלירים יכולים לפתוח כזה
ואז לשלוח ולקבל. ל- pipe הזה גם צריך לתת הרשאות
ולהגיד לו מי רשאי לקרוא ומי לרתיב.

גם ה pipes הם כמובן סוג של תקשורת ישירה.

במצגתי יש הרבה דוגמאות..

(11) 11.6.07
תרגול 05

היומולמד לתרגת sockets ב- unix.
הכל ב- unix הוא קבצים. כל פעולות הקלט ופלט עובדות בעזרת file descriptor. כמו שראינו למשל במקרה של ה- pipes.
אנחנו נדבר על שני סוגי פרוטוקולי אינטרנט.

TCP - בשבול לשלוח הודעות חיבים ליצור connection בין שני
הרתלירים. זה רמובן יהיה רצף system calls שיפתחו את הערוץ.
יש לו שני מאפיינים לוגיים: הוא עובד ב- FIFO והוא error-free.
הפרוטוקול לוקח את הרצף שאנחנו רוצים לשלוח ווחלק אותו ל- packets
כל חבולה יכולה לעבור במסלול שונה ברשת ⇐ כל packet יכל
להגיע בזמן כלשהו הפרוטוקול גם ידע לחבר את ה- packets חזרה
פרוטוקול מבטיח לנו שבסוף ההודעות יגיעו לפי סדר השליחה, כלומר
יש מרניזם ששומר על סדר ההודעות.

להבדול יש sockets שמאפשרים שליחת מידע שלא פעולה מקדימה של
פתיחת connection רק אומרים את המסן והו נשלחות אוטומטית.
ההודעות לא בטוח יגיעו וגם אין הבטחה שיגיעו לפי סדר השליחה.
אבל אם הן רבר מגיעות אז בטוח שאין בהן שגיאות. זה נקרא
UDP.

היומולמד תיכלס אבן עושים את כל הדברים האלה.
- socket מאפשר לתהלירים לתקשר דרך רשת. רתובת של socket
מורכבת מ- 6 בתים - IP זה ארבעה בתים, ו- port זה
עוד שניים.
יש struct בשם sockaddr_in שבו ממלאים את הרתובת.
כל packet מנותב עצמאית, לכן הרתובת צריכה להיות מובנת
לכל פלטפורמה שנמצאת בדרך של ה- packet. רפי שרולם יבינו
הרתובות צריכות להיות ב- Network Byte Order. זו פשוט מסכמה
לסדר של הבתים רפי שרולם יוכלו לפרש את המידע בצורה נכונה.

יש פוובציות שהופכות Host Byte Order ל- Network Byte Order
לאשל htons() או htonl(). ובחזרה ממירים ע"י ntohs() ו- ntohl().
בשבוע להיות פורטבוליים חייבים להמיר מידע ל- Network Byte Order
לפני ששולחים אותו ברשת.

socket descriptor הוא פשוט מספר socket() מקבלת את
התוקן (במקרה שלנו אינטרנט), סוג מידע ופרוטוקול. אם
הבריאה לפוובציה הצליחה פשוט מקבלים את המספר חזרה
אם אנחנו רוצים לקשר socket ל- port במרוגם שלנו
משתמשים ע"י הפקודה bind(). עושים את זה אם אנחנו מחשים
שרת שהולך להאזין listen() לקישורים שנכנסים ל- port
הזה.

לטרותים ידועים באינטרנט בדב יש port קבוע. למשל 80 זה
http, 23 זה telnet. כל הפורטים עד 1024 הם שמורים
מ- 1024 ועד 65535 אנחנו ירולים לבחור לשרות חדש כל שהו
שאנחנו רוצים. אבל אל תהליך אחר לא ירול להשתמש בו גם.

בשבוע להיתקשר משתמשים בפוובציה connect() שמקבלת מספר
של socket ואת הרתובת שאליה רוצים להתחבר (IP & port) הפוובציה מחזירה
ו- במקרה של כישלון וחשוב לבדוק את זה! זה קורה מהצד
של הלקוח.

השרת לעומת זאת אמור להאזין ללקוחות באמצעות הפקודה listen()
רשלקוח עושה connect(), עד שהשרת לא עושה accept() לקשר
הם נשמרים באיזה buffer. רשהשרת עושה accept() הוא יוצר
socket חדש שידבר עם הלקוח, ויחד איתו port חדש וזאת
כדי לפנות את ה port הראשי שמשמש רק לקבלת בקשות.

לסיכום: סרת פועל בכדל הפעולות הבאו:
- socket()
- bind()
- listen()
- accept()

ב-TCP: בשביל לשלוח הודעה משתמשים בפקודה send() שמחזירה
את מספר הבתים שאכן נשלחו מההודעה.
recv() היא הפוובדיה המתאימה שמקבלת הודעה וגם היא
מחזירה את מספר הבתים שנקלטו! - ס במקרה שמי ששולח
סגר את החיבור.

ב- UDP: ראן לא דריך את כל רדף הפקודות שיצרות רוברה.
יש פקודות sendto() ! - recvfrom() והן מקבלות את הכתובת
הרלוונטית הן עובדות בצורה דומה לפוב של TCP. recvfrom()
מקבלת פוינטר לכתובת של השולח והיא ממלאת אותו

סיכום של TCP:

Server	Client
- socket	- socket
- bind	- connect
- listen	- send/recv
- accept	
- send/recv	

המנגנון הזה היא blocking. רשאנחנו עושים accept() אין
את מי לקבל אז התהלך "חסם עד שהיה לקוח שרוצה לתקשר

סיכום של UDP:

listener	talker	
- socket	- socket	
- bind	- connect //	אופציונלי אז משתמשים ב- send ו recv רגיל
- recvfrom	- sendto	

יש כל מני פוובדיות שעוזרות לנו לפעוף מיד ע כל המכונה שאנחנו
מדברים איתה - שם, כתובת וכו' סיכום נמצא במצגת.

DNS - מתרגם כתובות שאנשים משתמשים בהן, למשל
www.notes-heaven.com לכתובת IP שלהם. על - port.
ספציפי ב- http ה- port היא תמיד 80. זה שירות
מפוזר אבל לא נדבר על פרטי המימוש שלו.

הפונקציה gethostbyname מקבלת שם ומחזירה struct hostent
שיש בו את כל המידע הדרוש אבל מה שמעניין אותנו הוא h_addr
שהוא הרתובת הבושנה ברשימת רתובות ה- IP זה טוב
שיש רמה רתובות רי זה מחלק בעצם את העומס בין מרונות
שונות ולכל אחת IP משלה

יש פונקציה שעושה מניפולציות על כל file descriptor. כרה אנחנו
ירנלים להפוך socket ל- non-blocking.

select מאפשרת לבבר בצורה סנכרונית רמה sockets.
יש אוסף של fd-ים לקריאה, אוסף לרתיבה ואוסף
יודאים מהכלל. אנחנו רוצים לדעת אם הקבוצות מורנות
לקריאה או רתיבה ואם היו exceptions ולכן זה מוגבל בזמן.
אחרי הזמן הזה הפונקציה תמיד תחזור אבל, למשל אם
מישהו מהקבוצה מוכן לקריאה אז היא גם תחזור והאמשנה
את האוסף אז יש לנו דרך לדעת מי מהם מדיוק מוכן לקריאה.

במלגת יש דוגמה לשמוש בפונקציה

(13) 20.6.07
ת. 50

Storage Allocation

היום נדבר על ניהול הזיכרון של תהליך בריצה.
תכנית זה איזה קובץ הרצה של הדיסק. בשבול להריץ אותה
צריך לטעון את מרחב הכתובות לזיכרון הראשי.
יש 3 זמנים שונים שבהם אפשר לתת כתובות לתהלך:
- בזמן קומפילציה - ראשר ל-*.ב נוצר ס.ל*.
- load time - זה בעצם זמן הלינק
- בזמן ריצה.

מתי אפשר לתת כתובת בזמן קומפילציה? ראשר יודעים מראש שנצטרך
את המשתנה הזה. אבל בשבול להריץ את התכנית שוב אז צריך
לקמפל שוב כי ירוץ להיות שרבר לא נקבל את אותו מרחב כתובות.
זה חוסרון רציני. אבל היתרון היא שזה חוסך בזמן ריצה.
גם בזמן הלינק היתרונות והחסרונות שיבים.
מה שמקובל היום הוא להקצות כתובות בזמן ריצה. אז לא
צריך לקמפל שוב. אבל גם פה יש בעיה גדולה. נניח שיש
context-switch. מי אמר שרשה תוכנית תצור לחץ היא תקבל
שוב את אותו מרחב כתובות?

- קומפיילר - הופך source ל- object והוא מסתכל כל פעם
רק על קובץ אחד
- לינקר - לקחת כמה object-ים ולהפוך אותם לקובץ הרצה.
- מעיה פעלה - טוען object לזיכרון ומאפשרת לכמה
תוכניות לרוץ באותו זמן

- כתובת לוגית - הכתובת הבאשונה היא 0 וזה ממשיך לפי
הגודל שהתכנית צריכה.
- כתובת פיסית - הכתובת האמיתית שרואה הזיכרון הבוש.

בין הרתובת הלוגית לפיסית תמיד יש איזשהו היסט קבוע.
MMU זה רכיב חומרה שדואג להמיר רתובות לוגיות לרתובות
פיסיות. הוא פשוט מוסיף את ההיסט שקבוע לכל תהליך (בהנחה
שהזיכרון שלו רציף ורק הוא רציף על המערכת).
במקרה של הקצאה רציפה אפשר להסתפק בשני רגיסטרים - אחד
שנמצא בו ההיסט הקבוע ואחד שיש לו את הגודל של
הזיכרון של התהליך - רדי שהוא לא יגלושממנו.
אז רשמקבלים כתובת לוגית משווים אותה מול ה. limit
רדי לבדוק שלא חרגנו מהמותר.

הבעיה הגדולה בהקצאה רציפה היא החורים שנוצרים
הזיכרון יוצא fragmented. נניח שהיו 3 תהליכים:

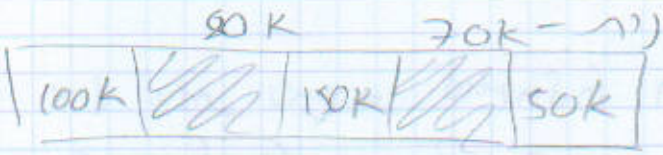
① 100	② 150	③ 200	פנוי 100k

ערשיו השני מסיים ובא תהליך חדש. אם הוא צריך פחות
מ- 150 אז אין לו שום בעיה אבל אם למשל הוא צריך
200k אז הוא לא יכול לקבל אותם למרות שתיאורטית
יש 350k פנויים.

הגיוני שאם בא תהליך חדש אז צריך לתת לו את החור הכי
קטן שמספיק לאכלס את התהליך. אבל זה עולה לנו בלחפש
את החור הזה - best-fit
גישה אחרת חוסכת זמן אבל מבזבזת זיכרון - עוברים על הזיכרון
ואת החור הראשון שמוצאים - נותנים לו. first-fit
וישגישה נוהפכנית שאומרת שאם נותן לתהליך את החור הכי
קטן האפשרי זה ישאיר שארית ובא קטנה ויש סיכוי מאוד
קטן שנוכל לאכלס שם תהליך נוסף ⇐ עדיף לשים דווקא
בחור הכי גדול. - worst fit.

יש מקרים שבהם best-fit יותר טוב מ- first-fit
ולהפך, במציאות, מתוך סימולציות ניבע שבד"כ
first-fit יותר טוב מ- best-fit. ושניהם יותר טובים
מ- worst fit. (טוב = כמה בקשות אפשר לשרת)
יש וריאציה של first-fit שלפיה לא מתחילים כל פעם
לחפש מחדש אלא מהמקום שבו עצרנו בפעם הקודמת.
להגם הגיוני כי אם עצרנו אפשהו זה אומר שלפני זה
לא מצאנו מקום ⇐ זה מקצר לנו את זמן החיפוש.

- פרגמנטציה חיצונית -



אם מגיע תהליך שדורש 100K אי אפשר לשרת אותו
כי כל אחד מהחורים בנפרד לא ירול לעזור אבל אם
מאחדים אותם זה מספיק.

- פרגמנטציה פנימית - מע' ההפעלה ירולה להקצות מקום
רק בבלוקים. אז אם התהליך של 100K השתמש רק
ב- 92.7K זאת פרגמנטציה פנימית שלו.

תהליך הוא יחידה אחת, אבל מבחינה לוגית אפשר הרבה
פעמים לחלק אותו לסגמנטים נפרדים - main, פונקציות,
אובייקטים, משתנים לוקליים/גלובליים, וכו'.
אז נשתדל שלפחות כל סגמנט רצה יהיה בצורה רציפה בזיכרון.
כל סגמנט יהיה רציף אבל הם לא בהכרח יהיו אחד אחרי השני.

ערשיו רבר לא מספיקים שני הרגיסטרים base ו- limit.
ערשיו צריך שניים ראלה לכל סגמנט. הרחובת הלוגית
ערשיו תהיה מספר הסגמנט + ההיסט בתוכו.
תהיה טבלת סגמנטים שאומרת את ה- base וה- limit
של כל סגמנט.

הגנה -

יש הרשאות שאומרות לאי מותר לגעת באיזה סגמנט. חוץ מזה לכל סגמנט יש validation bit שאומר אם הוא נמצא בזיכרון הבשי או לא. אם לא יוצא page fault ודריך להביא אותו מהדיסק.

לתכנית גם אפשר לעשות חלוקה לפי תפקידים. למשל, יש פקודות, משתנים לוקאליים, משתנים בבועים (שהם readonly).
ואפשר גם לעשות חלוקה לפי מה הוא readonly ומה הוא נותן לשינוי.

האם הקומפיילר ורול לתת כתובת לוגית לכל המשתנים בתוכנית?
א' - לא תמיד אפשר. למשל בהקצאה דינמית, או פונק' הקורסיביות
ב' - לא תמיד רוצים בכלל - כי אולי יש משתנים שאנחנו בכלל לא מגיעים אליהם, או שהם קיימים רק לשבריר שנייה.

זמן חיים של משתנה זה מתי שהוא מתזיק ערך ואנחנו ורגלים לגשת אליו. זמן החיים של משתנה גלובלי הוא כל התכנית ולהם אולי רצאי לגת מראש זיכרון. זמן החיים של משתנים לוקליים הוא זמן הרידה של הפלוק שהם חיים בו. לראלה אולי לא נרצה להקצות מראש.
חוץ מזה יש משתנים שמוקצים ע"י המשתמש באופן ישיר ע"י new או malloc ודריך לשחרר אותם עם free/delete.

- הקצאה סטטית - שמקצים מראש בזמן קומפילציה
- הקצאה דינמית - בזמן רידה עבור משתנים לוקאליים שחיים בזמן הפוקציה שהם נמצאים בה.
  - הקצאה על המחסנית (כל המשתנים ב- scope של הפרוצדורה)
  - הקצאה על הערמה (אם עושים x=malloc... אז x עדמו במחסנית והאובייקט בערמה) אלא אם כן הוא סטטי ואז הוא חי לכל אורך התכנית.

הזיכרון של המכונה מחולק ל-3 חלקים:
- מקום לאלפסיים והקוד של התכנית
- מחסנית
- ערמה

רשמריצים תהליך טוענים את מרחב הכתובות שלו לזיכרון הראשי.
הגדלים של כל הסגמנטים שמורים בקבצי object.
הקצאה דינמית היא מחוייבת המציאות כי יש כל מיני דברים
שאותנו לא יודעים מראש כמו כמה פעמים תרוץ פונק' רקורסיבית.

למשתנים יש 2 סוגי הקצאה.
- על המחסנית - זאת הקצאה הירכית. היתרון הגדול של
זה הוא הרציפות - כל המקומות הפנויים בחסנית הם ביחד.
- על הערמה - יש פון את אותן בעיות של פרגמנטציה
אבל בקטן. היתרון הגדול הוא הגמישות - אפשר להקצות
ולשחרר באיזה סדר שרוצים.
דריך לנהל free list שאומר איפה יש מקום פנוי ובאיזה גודל.
גם פה יש מה לדבר על best/first fit.
אפשר לייעל - מע' ההפעלה מקצה זיכרון בבלוקים. אז
אפשר לשמור Bit-Map שיש בו ביטים כמספר הבלוקים
וכל ביט אומר אם הבלוק המתאים פנוי או לא. אז אם
מתפשים ד' בלוקים פנויים דריך לתפש ד' ביטים דלוקים
כדופים. ואז גם לא דריך לשמור פוינטר לערמה
כי מיקום הביטים ב Bit-Map אומר לנו לאן דריך ללכת.

איך יודעים מתי אפשר לשחרר זיכרון? - השאין יותר אל תהליך
שמצביע אליו. למשל אם פותחים את אותו קובץ פעמיים אז
לא מעלים את כל המיד ע' הכלוונט לזיכרון אלא חולקים אותו. אז יש
reference-counter שמנהל ע"י מע' ההפעלה והיא דואגת לשחרר זיכרון
שאל ארד רבר לא מצביע אליו.

Garbage - Collection

זה נוהג נוח למתכנת אבל יש לזה סם חסרונות

- אילוצי זמן - במע' זמן אמת זה לא טוב כי אנחנו לא רוצים שפתאום בזמן לא מתוכנן הכל יפסק וזה ירוץ את ה GC.
- אילוצי זיכרון - הוא משחרר זיכרון מתי שהוא מחליט ולא השאנחנו רוצים.

איך GC עובד?

בשני שלבים:

- mark - עוברים על כל הפוינטרים שיש ומסמנים את מה שהם מצביעים עליו.
- sweep - לעבור על כל האובייקטים ולשחרר את אלה שלא מסומנים.

(16) 25.6.07
תרגול 05

מערכת הקבצים

מבחינת מע' ההפעלה קובץ כה פשוט רצף בתים שמאוגדים יחדיו.
משמעות הרצף מעניינת רק את האפלקציה שמשתמשתבו.
descriptor שלו זה מספר שמשמש אותנו לביצוע פעולות
על הקובץ. למעשה כה פוינטר לאיזו טבלה שבה יש את המידע
שדרוש לנו.

מה אנחנו רוצים לדעת על קובץ? למשל.
- למי שייך הקובץ - לצורך הרשאות לבעל הקובץ יש בדר אופציה
לעשות כל מה שבא לו.
- הרשאות
- זמן עדכון אחרון
- גודל הקובץ
- מיקום הקובץ - המידע עצמו - בדיסק

פעולות שרוצים לעשות על קובץ -
- לפתוח קובץ ולקבל את ה- fd שלו open
- לסגור - רשסיימנו לעבוד עם הקובץ close
- לבריאו - צריך להגיד לאן ובמה read
- לרתום לקובץ - צריך להגיד לו מה ולאן write
- להוסיף מידע לסוף קובץ append
- לבניז את המיקום הנורתי שלנו בקובץ seek
- לחזור לתחולת הקובץ rewind
- לשנות אטריבוטים של קובץ כמו הרשאות.

רשעושים fd=open("myfile", R) קורה הדבר הבא:

- מע' הקבצים בודקת את ה cwd ומוצאת ש "myfile" מיודע פנימית ע"י דו לא שלם. זה נמצא במן טבלה.
- משם אנחנו כולים לעבור לטבלה שיש בה את האטריביוטים של הקובץ. אם ההרשאות מתאימות אפשר משם ללרת להביא את הבלוק שבו נמצא הקובץ לזיכרון ואז fd מצביע למקום שבו הבלוק ימצא.

אנחנו מעבירים הכל לזיכרון כי גישה לדיסק לוקחת הרבה זמן ועדיף לנו לשלם קצת במקום בזיכרון מאשר לגשת לדיסק כל הזמן

לקובץ יש offset שמנוהל ע"י מע' ההפעלה, כל אופה שהיא מצביע משם אנחנו קוראים ולשם כותבים.

רשעושים read(fd, but, 100) fd מצביע למקום בטבלת הקבצים הפתוחים של ה kernel. שם אנחנו בודקים מהם הבלוקים שמרולים את המדע. מעבירים את הבלוק הרלוונטי לזיכרון, קוראים ממנו 100 בתים ומעדכנים את ה offset.

דבר דומה קורה בתיבה. מביאים את הבלוקים הרלוונטיים (יודעים איזה דריך כי ה offset ידוע, גודל הבלוקים קבוע ויודעים כמה רוצים לתיב) לדיסק, משנים את מה שדריך ומחזירים לדיסק. נשים ♡ שאם מגדילים את הקובץ, כלומר מוסיפים לו בלוק נוסף לא דריך להביא בלוק ריק מה דיסק, זה מיותר.

i-node זה הבלוק שמחזיק מידע על הקובץ. יש בו אטריביוטים ואח"כ פוינטרים לבלוקים של הקובץ על הדיסק.

היינו רוצים ש i-node יוכל להיכנס לבלוק אחד כי אנחנו מפלים אותו לזיכרון רשעושים פעולות על הקובץ. אבל זאת בעיה, כי כמות הפוינטרים שירולה להיות שמורה בבלוק אחד היא מוגבלת אז יוצא שכמות הבלוקים (כלומר גודל) בקובץ מוגבלת.

אבל מן הסתם אנחנו לא רוצים להגבול את גודל הקובץ.
אבוי! מה נעשה? אפשר לעשות משהו קצת חכם יותר.
אז לחלק מה פוינטרים ב- inode ובראו direct והם
ולבעיו ישירות לבלוקים של מידע בדיסק.
אח"כ יש פוינטר indirect שהוא מלביע לבלוק של מלבעים
ישירים. אז יש double indirect שמלביע לבלוק של
מלבעים indirect וכן הלאה עד triple-indirect.
ברור שעדיין גודל הקובץ מוגבל ע"ד, הוא מוגבל גם ע"י
גודל השדה "גודל קובץ" מאט ריבוטים. אבל זה פחות
מוגבל מה אופציה הקודמת. הדבר הזה הוא יקר כי עבור
קבצים גדולים אנחנו צריכים כמה גישות לדיסק, אבל ההנחה
הוא שרוב הקבצים שאנחנו עובדים איתם הם קטנים. בד"כ
יש 10 פוינטרים direct וזה מספיק.

לאז' הקבצים יש רשימה של inodes ורשתהליך צריך אחד אז
ה kernel ירול לחפש אחד פנוי. בשבול לשפר את הביצועים
ה- superblock מרול את מספר ה- inodes הפנויים.

אם כותבים לקובץ ונגמר הבלוק האחרון אז צריך להקצות בלוק
חדש מהדיסק. ב- superblock וש גם רשימה של בלוקים
הנויים (בעזרת bit-map) ושם אפשר למצוא את מה
שצריך. ברור שזה עובד יק בגדול שגודל הבלוקים הוא קבוע.

רשתהליך עושה פעולת I/O הוא נכנס לתנומה עד שמביאים
לו את המידע. מישהו צריך לעזור לו הוה טבעי להשוב שהמעבד
מביא את המידע מהדיסק לזיכרון. אבל אז המעבד עסוק
ואי אפשר להכלי תהליך אחר, וזה לא מה שאנחנו רוצים. בשבול
זה יש רכיב חומרה DMA שהתפקיד שלו הוא להביא בלוקים
מהדיסק ובסיום לעשות interrupt שגורם להחזרת התהליך
שלנו ל- ready queue.

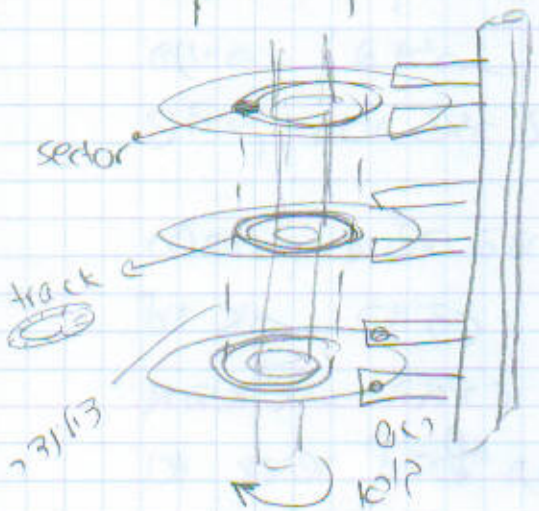
נניח שיש קובץ שכמה תהליכים קוראים ממנו. אין שום סיבה שה-
offset יהיה משותף לשני התהליכים שהרי ירכלת להיות
שכל אחד קובע רמות שונה. אז לכל אחד מהם דריך להיות offset
משלו. הרתיבה זה יותר מורכב. למשל אם יש כמה תהליכים שכותבים
ל- log משותף אנחנו רוצים שה- offset יהיה משותף.

לכל קובץ פתוח יש entry אחד לכל היותר בטבלת ה- inode.
בטבלת הקבצים הפתוחים יש entry לכל פתיחה של הקובץ.
וכרה לכל entry, כלומר לכל מי שפתח את הקובץ יכול להיות
offset משלו.

רמו ב, לכל תהליך יש טבלת filedescriptors שמצביעה
לטבלת הקבצים הפתוחים. ומשם הם מקבלים את ה- offset
אז אם יש כמה תהליכים נפרדים לגמרי יהיה להם offsets נפרדים.
ואם אנחנו רוצים שיהיו תהליכים עם אותו offset אפשר
מתהליך אחד להוליד ולדים ומרחב הכתובות משתכפל. לכן ה- fd
מצביע לאותו entry בטבלת הקבצים הפתוחים ולכן יש
להם אותו offset.

<u>מבנה הדיסק</u>

הדיסק הוא רדף לגוי של בלוקים. פיסיח זה לא קו ארוך.



ודיו לעבור בצורה סדרתית על הדיסק
עוברים על כל הסקטורים על צולנדר
ואז מגדילים את הצולנדר ועוברים עליו
זאת פעולה מבאנית ומה שהרי וקר צו
התזוזה של הבוש הקובץ => רוצים
למזע כמה שיותר את התזוזות
בין צולינדרים.

איך אפשר למצער את זה? אם יש רמה בבקשות I/O
אפשר לנסות לסדר אותן בסדר כזה רק שלא יהיו הרבה
תזוזות (כלומר ירול להיות שהבקשות לא ישורתו בסדר
שבו הן הגיעו)

ישרמה אופציות:

- FCFS - לשרת קודם את מי שבא קודם. זה כמובן ירול
לוקור ציבזגים גדולים בתזוזה וזה לא יעיל בכלל.
- SSTF - קודם לשרת את מי שה seek time שלו
הכי קצר. זה כמובן הרבה יותר יעיל מ- FCFS במקרה
הסטטי. אבל הרי אנחנו בינמיים וזה ירול להרעיב בקשות
קדשות שבאות.
- SCAN - הולכים רק בכיוון אחד - כל מי שבדרך משרתים
אותו רשמגיעים לסוף חוזרים אחורה ומשרתים את מי
שבדרך. זה גם יחסית הוגן וגם יחסית יעיל.
- אפשר לשפר את ההגינות אמנתייחס למעבר כמשהו מעגלי.
ברגע שנגיע לסוף ולק ישירות להתחלה. ראו זמן
ההמתנה לשירות היא אחיד (C-SCAN)
- C-LOOK - כמו C-SCAN אבל לא הולכים ממש עד
הקצוות של הדיסק אלא רק עד הבקשות הכי קיצוניות.
זה סיכון כי אם תבוא בקשה מיד אחרי שהגענו לקצה אנחנו
נפספס אותם כי רבר הזרנו. אז יש trade-off.

אמרנו שהדיסק מורכב מ-3 חלקים עיקרים - inodes, super block
ו- data. אפה שמים כל אחד מהם?
לא רדאי לשים את כל אלה פשוט אחד אחרי השני כי זה ויצור
הרבה seek. מה שקורה ב- unix היא ש- inodes והבלוקים
שבשורים אליהם נמצאים אחד ליד השני:

(19) 2.707
ת' OS

הייבודבר על ה API של מע' הקבצים. (פיתוח מלא במדגת)

טיפ חשוב: str error(errno) נותן תיאור מילולי של הבעיה.
perror מדפיס את השגיאה האחרונה למסך.

- lseek - נניח שמציצים את ה offset מעבר לגודל הקובץ. לכאורה זה מגדיל את הקובץ. אבל הלא באמת מגדיל אותו עד שאנחנו מחיש רותבים פנימה משהו ואז המרווח מתמלא באפסים.
- אין שום מניעה שהתזוזה בקובץ תהיה אחורה (offset שלילי) אבל אם זזים יותר מדי זו שגיאה.

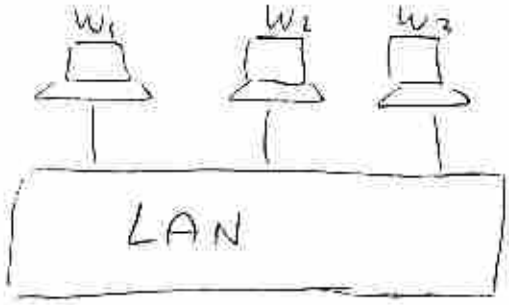
- read מחזירה במה בתים ובראו בפועל. הבתים שאבן נקבאו הם בהכרח ברצף מתחולת המקום שבו התחלנו לקרוא. אותו הדבר ב- write - הוא מחזירה כמה באמת נכתב וזה x הבתים הראשונים שבבשנו לרקוב.

- link - נניח שיש אתר בית של OS שנה הכאה אנחנו רוצים שנה יהיה בדיוק אותו אתר בית אבל אנחנו לא רוצים שהגישה תהיה דרך אותו מקום. אנחנו רוצים OS2006 ו- OS2007, אז אם נעשה link(OS2006,OS2007) ערשיו שני השמות ילביעו לאותו קובץ בדיוק. אז בטבלה של שמות הקבצים וה- inodes גם ל- OS2007 וגם ל OS2006 יהיה רקוב אותו מספר.
- ערשיו לקובץ יש כמה ליקים. אם אחד מהם נמחק זה לא אומר שהקובץ לא בייס יותר. רק כשכל הליקים נמחקים אז גם הקובץ נעלם.
- לעומת זאת יש symlink הוא יוצר קובץ חדש שבו יש את ה- path המלא של הקובץ המקורי. זה מסוכן כי כאו אם מוחקים את הקובץ המקורי אז הוא באמת נמחק. אבל זה נותן גמישות גדולה יותר כי אפשר להצביע לקבצים שנמצאים בעוד קבצים שונות מה שלא אפשרי בטרם link

(20) 11.7.07
ת' 05

Network File System

אנחנו נדבר על Local Area Network (LAN)
שנוחיבה אליה רמה מחשבים ואנחנו רוצים לידור להם מע'
קבצים משיתפת.



לכל אחת מהתחנות יש דיסק לושלה אבל אנחנו רוצים שיהיה
מע' משיתפת לרולם.

NFS זו מע' שנכתבר ע'י SUN היא משתמשת בפרוטיקול
UDP/IP. אחרי רמה זמן זה הפך לעשהו מאוד נפוץ ברשת.
המרניזה מאפשר למחשב לתפקד כשרת ולאפשר גישה מרחוק
לקבצים שפלו.

לכל תחנה יש מע' קבצים הירכית - הלוגיר אינשהו עץ. הקובץ עצמו
אולי הוא בכלל לא בדיסק הלוקאלי אלא אפשהו ברשת, אבל
זה שקוף לגמרי למשתמש. ושעושים פלו פעולות לא מעתיקים
את כל הקובץ לדיסק אלא שקבלים כל פעם את המלוק הדרוש
ע'י UDP.

יש שתי פעולות מרכיות. mount מעלה תיקיות מהרשת ל- view
הפרטי של משתמש ספציפי. רשמ"מאים ספריות זה לא שקוף.
צריך להגיד מאיפה בדיוק ואינו ספריה, אחרי ה mount אנחנו
יכולים לעבוד עם הקבצים ואילו הם בדיסק הלוקאלי.

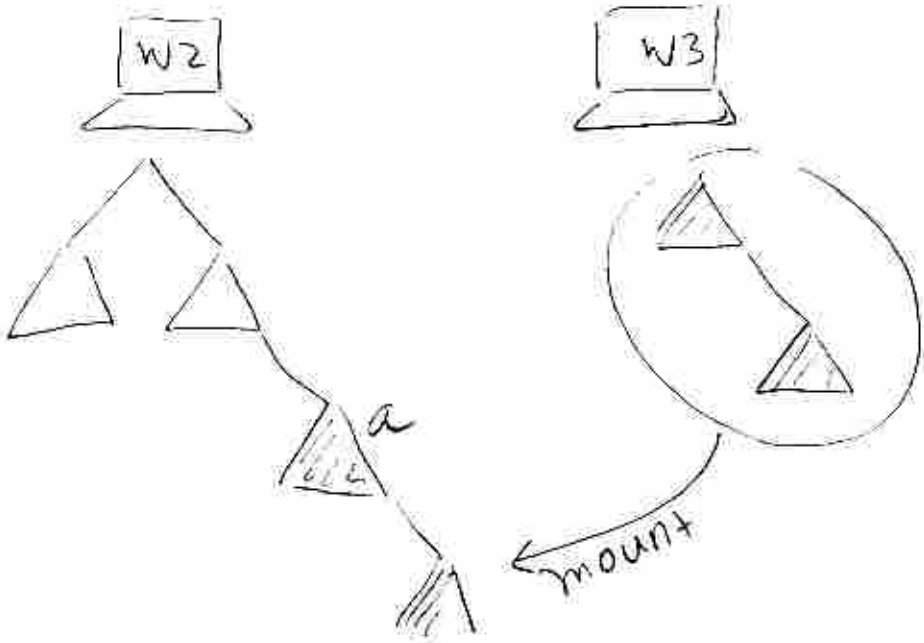
הרבה יותר קשה לכתוב מע' שתומרת במע' הטרוגניות. למשל מע'
הקבצים של יוניקס שונה מ של חלונות מה שעושים בדב' זה

איזה bridge - מין מרנה משותף לרולם. (כמו ה byte code של java).

ה NFS בנוי על אותה סמנטיקה של יוניקס.

אמרנו רבר ש- mount בונה על ספריות לכל תחנה וזה ועושה אישית לכל תחנה יק שירול להיות שלכל תחנה יהיה על אחר. היתרון משימוש ב- mount הוא הקונסיסטנטיות - כל שמות הקבצים אחרים.

דוגמה:



אחרי ה- mount הגישה של W2 לקבצים של W3 היא שקופה

ברור שיש בזה סניין של הרשאות - לכל תחנה יש רשימה של הקבצים שהיא ירולה לשתף

ונוח שבתבנו בלוק ל- a. זה קורה? יש שתי אפשרויות

- כל פעם שיש flush יש פעולה סינברונית שרותפת את הדולוק ל- W3
- אפשר לעשות מנגנון של cache.

NFS לא נותות כלים לסנכרן בין גישות לקבצים. כו האחריות של המשתמש.

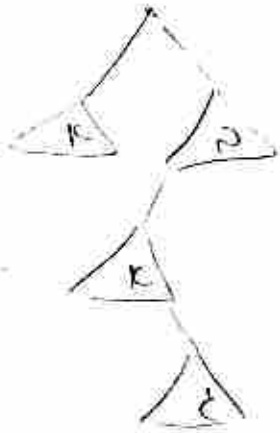
נניח שיש לקוח שנייש לשרת. ללקוח יש מלא שנשמר בדיסק
הלוקאלי של הלקוח ולא של השרת. אז אם השרת נופל
אפשר לעשות לו אתחול ובו אחד ירול להמשיך מאיפה שהוא הפסיק.
כלומר השרת הוא stateless. כל פעם הלקוח לריך לשלוח
את כל הפרמטרים לשות

בגלל שרוחב הפס ברשת הוא מוגבל ואולי הקבלים והספריות
ורוליב להיות מאוד גדולים אי אפשר לשלוח את הכל ביחד. אז
רשהשרת שולח לדהרים ללקוח הוא גם שולח משהו שאומר את
ומיקום שבו הפסקנו ואז הלקוח ירול לבקש שוב והוא יודע
מאיזו נקודה חסר לוי

אמרנו שאנחנו רוצים לתמוך ברשת הטרוגנית. אבל השמות
של הקבלים ירגלים להיות לא אחידים. אז לריך לתרגם את השמות
לפורמט של NFS. ותרגום הזה נעשה בלקוח עלמו ואז
הוא עושה את הבקשה

נשירו לא ש- mount הוא לא טרנזיטיבי. כלומר אם a
עושה mount ל-b ו- b עשה בידם mount ל-c
אז a וקבל את הקבלים הלוקליים של b ולא את c.

הוא מזה יש פה עניין של cascade mount
העל ירול להיות בנוי רב שהמבורות של הקבלים נעורבבה
זה עם זה. כלומר, זה לא שכל ערל הוא מוחשב ספליים.



אז לריך לשמור את
המיקומים של הקבלים
מיבשהו.

11.7.04
תחישוביות

(*) $L_1 = \{ \langle M \rangle : L(M) = L(M')$ קיימת M' עם 2006 מלבנים רק ש-

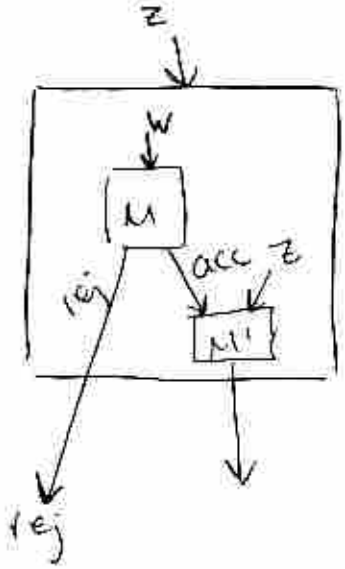
נראה ש- $L_1 \notin RF$.

נראה רדוקציה $\overline{A_{TM}} \leq L_1$

משיחולי טפירה, קיימת מ"ט M' שמזהה שפה שלא ניתן לזהות

מפרחות מ- 2006 מלבנים

בהינתן $\langle M \rangle, w$ נגדיר M_1 באופן הבא:



מתקיים $w \notin L(M) \Leftrightarrow M_1 \in L_1$

(שהרי $L(M_1) = \emptyset$)

(*) $L = \{ (01)^n 0 (1010)^n : n \geq 0 \}$

נשים ל שמתקיים $L = (010101)^* 0$

כלומר כל המילים מהצורה 0^*101 שבהן מספר

ה- 1-ים מתחלק ב- 3, זה בוודאי רגולרי

$L = \{ (01)^n 0 (1011)^n : n \geq 0 \}$

אינה רגולרית. לוקחים בלמת הניפוח את $(01)^p 0 (1011)^p$.