סיכומי הרצאות הקורס "מערכות הפעלה"

אוניברסיטת תל אביב, סמסטר א' 08/09

מעודכן לתאריך: 9.2.09

איך ללמוד מהסיכום?

הסיכום אינו מהווה תחליף למצגות, ויש לקרוא אותו בנוסף למצגות, המופיעות באתר הקורס.
מומלץ לקרוא את הסיכום לצד המצגות. אם הדבר לא נוח, התחילו עם הסיכום ולאחר מכן עברו על המצגת (אך אם משהו לא מובן בסיכום, עברו למצגת לבדוק את הנקודה).
מומלץ לקרוא את הסיכום בגרסה אלקטרונית ולא להדפיסו. במידה שהודפס – מומלץ לנצל את הדפים פעם נוספת (על ידי שימוש בצד השני שלהם לדוגמא).

שיעור 1

מצגת Introduction I

ניתן לקרוא את סיכום ההרצאה בנפרד. דברים מאוד כללים על הקורס ומה זה מערכת הפעלה.

שיעור 2

מצגת Introduction II

יש להבדיל בין מערכת ההפעלה לבין הכלים הנוספים שיש מערכות הפעלה שמצרפות למערכות הפעלה שלהן (כמו קומפיילרים, סרברים ולמיניהם, GUI וכו')

<u>מה זה יוניקס</u>?
יש סטנדרט שכל מערכות יוניקס צריכות לעמוד בו – זוהי למעשה קב' מערכות הפעלה עם ממשק משותף שנוצרו על ידי יצרנים שונים שהסכימו על הסטנדרט (דוגמאות: סולאריס, לינוקס, BSD, דוגמאות נוספות בשקופית 12), וכך ניתן לכתוב תוכניות שמתאימות לכולם (יש תקן).

מערכות הפעלה חינמיות כמו לינוקס הן לא באמת יוניקס, כי הן לא הלכו למבחן יוניקס שבודק אם אתה תואם לתקן posix. למה? כי זה עולה כסף. אצלנו מתייחסים אליהם ככאלו, כי הם תכל'ס עומדים בתקן. השם יוניקס הוא ירידה על MULTIX.

שקף 11 – הרחבה על היסטורית יוניקס. יש גבול לכל תעלול.

<u>GNU</u>
ראשי תיבות של: "Gnu's not UNIX". כן כן, ראשי תיבות רקורסיביים.
סט של כלים חופשיים שפותחו על ידי קב' ה-FSF, free software foundation, שהם כותבים קוד יוניקס ומפיצים אותו לעולם.
רוב מערכות ההפעלה כשאנו עובדים עם לינוקס, אנו אוטו' משתמשים בכלים של GNU (כלומר כל מה שמעל הלינוקס של GNU.

יש רישיון לשימו שתוכנה שנקרא GNU GPL, שהוא אחד הרישיונות החופשיים הנפוצים מעולם – מאפשר לשכפל קוד, להפיץ אותו מחדש וכו', תחת מגבלות. עוד דוגמאות לרישיונות חופשיים – שקף 14. יש חופשיים יותר ופחות.

חשוב להבדיל בין חופשי לחינמי –
Free Beer – מקבלים חינם, כמו תוכנות רבות, כמו Opera.
Free Speech – חופשי ברמה שניתן לשנות את הקוד, להפיץ מחדש וכו'.

מה זה **<u>הפצת לינוקס</u>**?
לינוקס זה רק הקרנל, ואי אפשר לעבוד רק איתו. לקחו הרבה כלים יחד: אופן אופיס, GNU וכו', שמו יחד, תוכנית התקנה שתתקין הכל יחד וקראו לזה 'הפצה'. כל הפצות הלינוקס משתמשות בלינוקס ובכלים של GNU. ההבדלים בין ההפצות לא מאוד גדולים – מנהלי חבילות שונים וכו'.

בקורס נשתמש בהפצה: Fedora Core 9.

<u>יש להבדיל בין פונ' ספריה לפונ' מערכת הפעלה</u>:

- <u>פונ' ספריה</u> – להן אנו באמת קוראים, זה קוד של GNU, והוא קורא לקוד של לינוקס.

1. <u>דוגמא בשקף 22</u> – כשאנו כותבים קוד C ורוצים לפתוח קובץ, אנו משתמשים בפונ' fopen, שהיא חלק מ-GNU. והיא קוראת לפונ' open שהיא פונ' של לינוקס (פונ' מערכת הפעלה). אנו בדרך כלל לא קוראים ישירות לפונ' של לינוקס.

2. <u>ההבדלים בין פונ' ספריה ופונ' מערכת הפעלה</u> – המרחב שפועלים בו. פונ' ספריה בדר"כ לא תלויות מערכת.
   - <u>Man 2</u> – פונ' של לינוקס.
   - <u>Man 3</u> – פונ' ספריה.

שקף 21 – משווה בין פונ' ספריה לפונ' OS, מבחינת נפיצות באגים, יעילות וכו'.

OS function	Library function
Lowest level interface	Usually calls OS function
Section 2 of man	Section 3 of man
System dependent and may not exist on different systems (or used with different name or parameters)	Programming language dependent (but not system dependent)
Almost always bug free	Almost always bugfree
Almost always efficient	Almost always efficient
Usually – kernel space code	Usually user space code

<u>יוניקס בנויה על שני רעיונות מרכזיים:</u>

תהליך וקובץ.

כל שירותי מערכת ההפעלה ממומשים או כקובץ או כפרוסס, תהליך.

לדוגמא: Logger של המערכת הוא פרוסס, אבל סוקט זה קובץ.

מצגת OS Development Cycles

זיוני שכל, אין צורך לסכם. חזרה על שיעור ראשון. אנשים יקרים.

שיעור 3

מצגת The Unix Process

מה יש לתהליך?

- זיכרון שלו (של הקוד ושל הנתונים). (התוכנה כתובה על הארד דיסק. כשמריצים אותה היא מועתקת לזיכרון, והיא רצה משם. אז למעשה התוכנה תופסת כמה חלקים בזיכרון – גם את החלק של הקוד שהעתקנו וגם עוד חלקים אחרים מהזכרון בשביל לשמור את הנתונים).
- משתנה סביבה – להבין מה זה.
- טבלת File Descriptors
- Signal handler

מה זה סיגנל? (שקף 4)
אותות, הדרך של המחשב להודיע על software interrupt.
דוגמא לאותות: "תמות", חלוקה ב-0, segmentation violation, הבן שלך מת.
נגיד לחיצה על X בוורד היא שולחת למערכת הפעלה, וה-OS שולחת לוורד שייסגר. חצי מהאותות מסתיימים בצורה טראגית..

<u>סיגנלים מעניינים מתוך המצגת</u>: שקף 5

- please shutdown – Sigabrt/Sigterm (זה מה שנשלח לו כשלוחצים על X)
  1. תיסגר יפה. נותן לו זמן לשמור את הנתונים שלו, לגבות דברים.
- Sigkill – אי אפשר לתפוס אותו באמת, אתה פשוט מת.
  1. ממש הורג, חותך באותו רגע, מעיף נתונים מהזכרון. מאבד מידע.
  2. מערכת ההפעלה דואגת לשחרר את כל הזכרון, לעשות free בשבילנו. (נדבר על זכרון בשיעור אחר)
- Sighup – נפוץ בעיקר בשימוש בשרתים. נגיד שיניתי הגדרה בשרת ואני רוצה לטעון את ההגדרה החדשה. אם אני אכבה את השרת ואדליק, בין לבין הוא לא יתן שירות. אז אני משתמש בסינגל הזה, כנראה גורם לו לקרוא מחדש את הפרמטרים. אם אתה הופעלת מתוך טרמינל, והוא נסגר, אתה מקבל את האות הזה ובכך מסיים את חייך. לשרתים (דיאמון) אין טרמינל מעליהם ולכן לא יקבלו סיגנל זה כי הטרמינל מעליהם מת, אלא כי רוצים שהם יקראו את ההגדרות שלהם מחדש (זה הסכם מקובל שדיאמון קורא את ההגדרות מחדש כאשר הוא מקבל את הסיגנל הזה).
- Sigsusp – עושה suspend לתוכנה (אי אפשר לתפוס אותו)
- SIGCHLD (sigcld) – הילד שלך מת.
- ה-OS מחזירה תשובה ל-select באמצעות אות
- ה-OS מחזירה תשובה ל-wait באמצעות אות.
- גם ALARM – אותו דברץ.
- sigUSRx (x הוא 1 או 2) – לשימוש המתכנת. המתכנת יכול להגדיר מה הסיגנל הזה יעשה.

סביבה – שקף 7
רשימה של פרמטרים שעוברים בתורשה בין תהליכים.
~1

<u>זכרון</u>
יש לתוכנה כמה אזורים שהיא יכולה לשמור בהם נתונים שלה:
STACK – מחסנית, זה נגיד השיטה שקרואים לפונ'. רצה על קוד, קוראת לפונ'. צריכה לזכור מאיפה קראתי לה כדי לרוץ ולהמשיך לרוץ כדי לסיים. שומר על ה-STACK איפה הייתי עכשיו. זה בדיוק הרעיון של STACK -

כל פעם יוציא מ-STACK את האחרון (רקורסיות – הכל יודע מתי לחזור כי ב-STACK נשמר ה-trace של התהליך).
ב-STACK שומרים כל מיני דברים – גם נתונים, נגיד int a=5. כל דבר שלא עשינו לו malloc הוא אוט' על ה-STACK.
כל מה שעשינו לו malloc הוא על ה-HEAP.

ב-C אין garbage collector כמו ב-JAVA, צריך לעשות free למשתנים שהגדרנו על ה-HEAP. משתנים שאנו מגדירים על ה-STACK לא צריך לעשות FREE. הם אוט' משתחררים לפי ה-scope.

- malloc עושה הקצאת זכרון דינמית על ה-HEAP.
- Alloca – עושה הקצאת זכרון על ה-stack.
  1. ההבדל המהותי – ב-alloca אני לא צריך לשחרר את הזכרון.

<u>לכל התהליכים יש אבא</u>
האב הקדמון הוא **init**.
כלומר – כל תהליך נוצר על ידי תהליך אחר. אחים נמצאים באותה קב' תהליכים, ואפשר גם לשלוח סיגנלים לקב'. למה זה טוב? נגיד שאני שרת אינטרנט, יש לי המון בנים וכל אחד מדבר בבנאדם אחר – אני רוצה לשלוח לכולם יחד הודעה שיפסיקו את הטיפול.

Getpid() – get process ID – מחזיר את ה-PID שלי.
Getppid() – get parent process ID – מביא את ה-PID של אבא שלי.
קשה לדעת מה ה-ID של הבנים שלך, יש לך רק הזדמנות אחת והיא ב-FORK. אחריות של האבא לשמור את השמות של הילדים שלו. אין "סיסטם קול" שמאפשר לקבל את הבנים שלי.

Daemon
תהליך שרץ ברקע כל הזמן, ונותן שירותים עבור המשתמש.
הדוגמא הכי נפוצה זה שרת שרץ ברקע על המחשב.

כדי להפוך לדמון צריך להרוג את אבא ולהיות מאומץ על ידי init. אפשר לדוגמא לשלוח KILL ל-PID.

<u>דוגמאות ל-daemon:</u>
פחות או יותר כל מה שנגמר באות d.לדוגמא httpd, sshd.
Syslog – דוגמא ל-daemon. תהליך שיודע לכתוב לוגים, ותוכנות מתקשרות איתה. היא תמיד ברקע, וכל הזמן תוכנות יכולות להגיד לה לרשום את הלוג.
פונקציות שמשתמשים בהן כדי לתקשר עם ה-syslog:

- Openlog (3)
- Syslog (3)
- Closelog (3)

אלו פונ' ספריה ולא פונ' מערכת. להיזהר, יש גם syslog(1), כלומר כתיבה ב-shell.

דוגמא לסיסלוג: (נצר ניתק את המודם וחיבר).

```
Nov 20 04:57:39  89-138-166-80 login[16146]:  USER  PROCESS: 16146  ttys000
Nov 20 04:59:35  89-138-166-80 pppd[17475]:  Connection terminated.
Nov 20 04:59:36  89-138-166-80 pppd[17475]:  PPTP disconnecting...
Nov 20 04:59:36  89-138-166-80 pppd[17475]:  PPTP disconnected
Nov 20 04:59:37  Macintosh configd[14]: setting hostname to "Macintosh.local"
Nov 20 04:59:38  Macintosh pppd[16479]:  pppd 2.4.2  (Apple version 314) started
    by root, uid 501
Nov 20 04:59:38  Macintosh pppd[16479]:  PPTP connecting to server
    '172.26.255.245'  (172.26.255.245)...
Nov 20 04:59:38  Macintosh pppd[16479]:  PPTP connection established.
Nov 20 04:59:38  Macintosh pppd[16479]:  Connect: ppp0 <--> socket[34:17]
Nov 20 04:59:38  Macintosh pppd[16479]:  PAP authentication succeeded
Nov 20 04:59:38  Macintosh pppd[16479]:  local  IP address 89.138.196.228
Nov 20 04:59:38  Macintosh pppd[16479]:  remote IP address 212.143.205.162
```

שקף 13 – איך משתמשים בפונ' סיסלוג.

```
void    syslog(int priority, const char *message, ...);
void    openlog(const char *ident, int logopt, int facility);
void   closelog(void);
```

חשוב לשים לב שהלוגים נשמרים בתיקייה:

/var/log/

בדר"כ בקובץ messages.
הרבה לוגים אח"כ נדחסים ונשמרים. בסוף כל יום יש log rotate, פונ' דוחפת את הלוגים הצדה, נדחים וכו' – ופותחים לוג חדש.

בעבר רצו בלינוקס המוני דמונים, וזה גרם לשכפול קוד (כי הרבה מהם עשו אותו דבר אבל קצת שונה). הגיעו למסקנה שזה לא טוב בגלל שכפול באגים.
במקום זה יש עכשיו דמון שמקשיב במאות סוקטים באמצעות סלקט, במקום שהיו מאות דיאמונים. רק כשיש בקשה הוא יעשה fork. הדמון הזה נקרא inetd super-server.

מצגת Multi-process-Multi Threaded

<u>רעיונות בסיסים</u>

פרוסס הוא יישום - וורד, שרת אינטרנט, קובץ EXE. פורמלית : תוכנית עצמאית שיש לה מיין והגנת זכרון מתוכניות אחרות.

ת'רד - יישומים עצמאיים שנוצרים על ידי פרוססים אחרים. נגיד שרת ווב - מחכה לבקשות, רוצה לטפל בכמה בקשות במקביל, לכל בקשה פותח ת'רד.
כלומר מיני תוכנית, תוכנית בתוך תוכנית, שהיא רצה במקביל לתוכנית הראשית (שזה הפרוסס-יס).
בתוכנית אני יוצר הרבה מיני תוכניות - הן לא מוגנות זיכרון בינן לבין עצמן (חסרון ויתרון ביחד).
הרעיון הכללי - לעשות דברים במקביל. (יתרונות במצגת - לענות לכמה אנשים ביחד, לעשות דברים בזמן שכותבים וכו').

פרוסס יחיד - זה מה שהיה פעם בהתחלה. (זוכרים את DOS..?)
סלקט - כשרוצים פרוסס יחיד, אבל עם אפשרות של המתנה לכמה דברים ביחד. נקרא select - פונ' של לינוקס (כלומר ב- man היא נמצאת ב- 2).

ה-file descriptor - נגיד התכנה שלי פותחת קבצים, הקבצים שפתוחים ה-FD הוא כמו פוינטר לקבצים הפתוחים, סמן, מספר, int.
בלינוקס כל דבר הוא קובץ, גם התחברות לשרת. כשאני כותב לקובץ שמסמל בעצם את ההתחברות מול השרת - מה שבעצם נעשה זה שמידע נשלח לשרת. גם המסך הוא קובץ בלינוקס.

ה-API של select(2): (השתיים בסוגריים מסמל שזו פונקצית מערכת ולא ספריה)
המטרה של סלקט - אם אני רוצה לנטר כמה FD.
דוגמא לשימוש בסלקט: כשעושים scanf ב-C - תוכנה קופאת עד שקלט לא מתקבל. אנחנו רוצים לחכות 10 שניות ואם לא מתקבל - להמשיך.
אפשר לעשות select על FD שמתאר standard input.
רק בשביל לוודא: סלקט זו פונ' שמריצים בקוד (ממש כותבים אותה בקוד C שלנו).

עוד דוגמא לסלקט -
כששרת מחכה להתחברות, הוא קופא. כשאנו רוצים לחכות להתחבריות מכמה מקומות. למקבל. סלקט אומר -
אלו הדברים שאני רוצה לנטר, תודיע לי כשאחד מהם עושה משהו.
סלקט מקבל **ארגומנטים** :
1. ה-FD המקסימלי + 1 (נגיד יש שלושה FD, שהמספרים שלהם זה 15,17,34. אני אשלח כאן 35 - כדי שידע עד איפה לרוץ)
2. לאיזה FD אני רוצה לשים לב לאלה שאפשר לקרוא מהם (רשימה של FD, למעשה אובייקט שנקרא fd_set – מבנה קיים, שבו יש פונ': FD-SET וכו').
3. לאיזה FD אני רוצה שאפשר לכתוב עליהם שכותבים אליהם. (אנחנו לא ניתקל בזה ולא נשתמש בזה)
4. לאיזה FD אני רוצה לשים לב שיש בהם אקספשן אליהם. (לדוגמא האם קרתה התנתקות בסוקט – הגדרה בהמשך, דיסק ברשת - נעלם)
5. אובייקט Time Out - אחרי כמה זמן להפסיק לנטר.

סלקט חוזר (עד שהפונ' סלקט לא חוזרת התוכנית לא ממשיכה) אחרי שאחד מתרחש: או שמשהו מתרחש בקבצים, או שמתרחש time out. כלומר סלקט הוא פונקציה Blocking.
סלקט הורס את הרשימה, וצריך כל פעם ליצור אותה מחדש.
0 - ה-descriptor file של ה-standrad input.

בעצם למה בכלל יש את הרעיון של סלקט?
לא רוצים שכל תהליך יבדוק לדוגמא אם הוא קיבל קלט, אלא ש-OS תודיע לו כשזה קורא. שלא כל פעם ישאל: קיבלתי? קיבלתי?

<u>סלקט לא באמת מחזיר</u> - אלא הורס את ה-set ומחזיר במקומו את התוצאה. כשסלקט חוזר זה אומר שמשהו קרה. איך יודעים מה ממה שביקשנו קרה?
השיטה לבדוק היא באמצעות בדיקה של הרשימה ששלחתי (select שינה אותה). את זה אני עושה באמצעות הפונ' FD_ISSET (אני שולח לה FD שרוצה לבדוק (מתוך מבנה הנתונים), ואת הרשימה ששלחתי לו. (שקף של select example)

דוגמא לשימוש בתרדים: יישומים עצמאיים שנוצרים על ידי פרוססים אחרים. נגיד אפאצ'י - מחכה לבקשות, רוצה לטפל בכמה בקשות במקביל, לכל בקשה פותח ת'רד.
תרד מיני תוכנית, תוכנית בתוך תוכנית, שהיא רצה במקביל לתוכנית הראשית (שזה הפרוסס-יס).
בתוכנית אני יוצר הרבה מיני תוכניות - הן לא מוגנות זיכרון בינן לבין עצמן (חסרון ויתרון ביחד).
הרעיון הכללי - לעשות דברים במקביל. (יתרונות במצגת - לענות לכמה אנשים ביחד, לעשות דברים בזמן שכותבים וכו').

בעיות עם סלקט
- אפשר לנטר רק דברים שהם FD (מיוטקס וחישוב לדוגמא אינם FD – ועל כן אי אפשר לוודא מתי הוא הסתיים). סוקטים וקבצים כן עובדים בשיטה של FD, וגם אינפוטים של מסך.
- הוא לא הוגן – כלומר אם הבאתי לו מלא דברים לנטר, הוא לא באמת עובד בצורה הוגנת. יכול להיות שאלו עם FD הקטן יקבלו יתרון בזמן דיווח (כי אין באמת דבר כזה מקבילי במחשב, סריקה רציפה).
- הורס את הארגומנטים שלו – כל פעם צריך ליצור את "הרשימה" של ה-FD מחדש.
- יש כבר פונ' חדש ב-UNIX, פול (POLL) שאנו לא עובדים איתה. (והוא לא באמת זמין בכל מקום).

מתי להשתמש בסלקט(2)?
1. כשיש קלטים מרובים – תודיעי לי אם אני מקבל מאחד מהם (ילחץ עכבר, אנטר וכו')
2. (הסיבה שאנחנו לא משתמשים עדיין בתרד אלא בסלקט) ניתן לטפל בקלטים בצורה סדרתית – הסלקט לא מאפשר לנו לטפל בהרבה דברים בו"ז, רק לוודא אם קיבלנו משהו בו"ז. ורק אז אני הולך ומטפל בו באמצעי אחר.
3. טיפול בבקשה יחידה היא מהירה
4. כאשר כל האינפוטים הם FD
5. רוצים להימנע מלעבוד עם ת'רדים. רוצים שיהיה אווירה של מיקבול מבלי להיכנס לבעיות של הת'רדים.

במה לא להשתמש במקום סלקט?
1. API -ים ישנים
2. busy waiting
3. לא להשתמש בת'רד כשאפשר לעשות זאת באמצעות סלקט.

tasks multiple

הגדרות:

1. פרוסס (תהליך) – תוכנית רצה. תוכנה שיש לה מיין, רץ שורה אחרי שורה.
2. ת'רד (חוט) - יישומים עצמאיים שנוצרים על ידי פרוססים אחרים. נגיד אפאצ'י - מחכה לבקשות, רוצה לטפל בכמה בקשות במקביל, לכל בקשה פותח ת'רד.
    1. כלומר מיני תוכנית, תוכנית בתוך תוכנית, שהיא רצה במקביל לתוכנית הראשית (שזה הפרוסס-יס). בתוכנית אני יוצר הרבה מיני תוכניות - הן לא מוגנות זיכרון בינן לבין עצמן (חסרון ויתרון ביחד). הרעיון הכללי - לעשות דברים במקיל. (יתרונות במצגת - לענות לכמה אנשים ביחד, לעשות דברים בזמן שכותבים וכו').

טסק (משימה) – שם כולל לשניהם.

כולם noun. מי שחושב שזה פועל – צריך להרוג אותו. הוא יותר גרוע מרוצח ואנס ילדים. לכל הפחות, צריך לתת לו 80 מלקות.(??)

מולטי טסקינג – שתי משימות ירוצו אחת ליד השני (לא היה פעם בדוס). לקרוא מייל ולכתוב בוורד בו"ז. לטפל בשתי בקשות במקביל וכו'.

שיטות שונות:

1. פעם – graceful multi – כל אחד מקבל את המעבד בתורו, כמה זמן שהוא צריך עד שהוא מעביר אותו הלאה, ואז זה עובר הלאה. ככה עבדה Windows 3.11. הבעיה היא שאם תכנית נתקעת אז היא לא מעבירה את המעבד ואז כל המערכת נתקעת...
2. Pre-emptive multi – יש קרנל של מערכת הפעלה שמחליט מתי לתת צומי למשימה ומתי להפסיק לתת לו – הוא מחלק את המשימות. זה מה שתכל'ס עושים.
    a. Pre-empt – הפעולה של להחליף בין פרוססים שרצים.
    b. Scheduler – החלק במערכת ההפעלה שאחראי על הניהול.

איך זה עובד בפועל?

מריץ pre-emptive, לכל פרוסס יש:

1. זכרון
2. מחסנית
3. משתנים גלובליים
4. האוגרים בתוך ה-CPU נבדלים. ברגע שאני רוצה להביא תוכנה אחרת, צריך להחליף בין האוגרים, לשים את האוגרים של התוכנה האחרת. רק במחשבים החדשים יש שני מעבדים (לא על זה אנחנו מדברים). ***

איך מתקשרים בין יישומים?

תקשורת ביניהם – יש דרכים רבות, נלמד שתיים: TCP, UDS.

איך יוצרים פרוסס חדש?

באמצעות הפונקציה **FORK**

מה פורק עושה? משכפלת את התוכנית. כל המשתנים, זכרון וכו' משוכפלים. ברגע שאנחנו מגיעים לפונקציה הזו.

איך נבדיל מי האבא ומי הבן? (האבא – ממנו שכפלתי, הבן זה מה ששוכפל, החדש).

הפונ' FORK מחזירה לאבא את ה-PID של הבן.

הפונ' FORK מחזירה לבן 0. (PID שונה מ-0).

דוגמא:

```
if (!fork()) { // this is the child process
                close(sockfd); // child doesn't need the listener
                if (send(new_fd, "Hello, world!\n", 14, 0) == -1)
                    perror("send");

                close(new_fd);
                exit(0);
            }
            close(new_fd);  // parent doesn't need this
```

הפונ' FORK רצה משכפלת את הכל. לאבא היא מחזירה PID של בן. האבא מדלג על ה-IF הראשון במקרה הזה. בקטע הזה הילד הורג את עצמו (FORK מחזיר 0 אם זה הבן).

יש דרכים במקום FORK – execXXX – משפחה של פונ' שעושה את אותו הדבר, רק עם פרמטרים (ל-FORK אין פרמטרים, זה אם אני רוצה משהו יותר מתוחכם). פרק 8.10
המשפחה הזו היא מחליפה אותנו (כפרוססים) בפרוסס אחר שנקרא מתוך הדיסק. כלומר, נניח אני קורא ל – LS, אז מה עושה ה- ?SHELL היא קוראת לפורק ובכך משכפלת את עצמה, האבא קורא לל- WAIT ובכך מחכה שהילד ימות, והילד קורא ל- EXEC ומחליף את עצמו עם LS אותה רצינו להריץ.
כש- LS מסתיים הילד ימות, והאבא יתעורר (עקב קבלת SIGCHLD).

<u>חידה לשימוש ב-FORK – (שקף 15)</u>
יש פוס' שמדפיסה HW, משתכפלת, מדפיסה אנטר, ואז עושה FLUSH ל-STDOUT. לכאורה רק האנטר אמור להיות מודפס פעמיים.
מה קורה באמת? כשרושמים printf, זה נכנס לבאפר, ועד שלא עושים לו flush זה לא נרשם למסך. ואז כשעושים fork זה גם משכפל את הבאפר.

<u>איך להעביר מידע בין פרוססים</u>?
יש מלא דרכים:

- תקשורת רשת
- UDS (פרוטוקול שהמציאו בלינוקס, יותר מהיר מתקשורת אינטרנט)
- דרכים נוספות במצגת.

אחרי שהאבא יצר את הבן שלו, הוא יכול לחכות שהוא ימות. כשהפורסס מת הוא מחזר קוד חזרה.
איך הוא מחכה?
באמצעות פונקציה WAIT, או ()waitpid.
שהפרוסס ילד שלנו מת, הוא שולח סיגנל SIGCHLD (למעשה מ"ה שולחת אותו), ואם לא עשו לו WAIT (רק אבא שלו יכול לעשות את זה), אז הוא נהיה זומבי. זומבי זה בעצם הערך שמוחזר מהמיין של הילד.
ילד זומבי בסוף יאומץ על ידי התהליך init, (שתמיד עושה WAIT לילדים שלו) כשאבא מת, INIT מאמץ את הילד. בכך INIT מעלים בסופו של דבר את הזומבים.

<u>דרכי תקשורת</u>:

- תקשורת רשת – לקרוא מדריך של ביג' (מה הוא שקע, מבנים וקריאות).

<u>בעיות עם מולטי – פרוסס-יס</u>

1. ניהול מידע ביניהם – נגיד יש משאב משותף ששניהם משתמשים בו זה בעיה.
2. מגבלות API – כשעושים FORK ורוצים לחכות שהילד ימות (WAIT), ורוצים במקביל להקשיב ל-socket עם סלקט. ה-API לא תומך בזה, כי לפעולה של מוות של הילד אין FD.

3. החלפת תהליך היא יקרה.

Context switching

המנגנון שמחליף בין פרוססים. חלק ממערכת ההפעלה.
יש מעבד אחד – מסוגל להריץ רק פרוסס אחד בזמן אחד. אם רוצה להריץ הרבה פרוססים – לא במאת יכול, עושה טריק. כל אחד עושה קצת. מערכת ההפעלה אחראית על כמה שכל אחד עושה. אומרת : עכשיו אתה מקבל את המעבד. מי שמחליט – זה ה-scheduler. ה-context switch – מחליף את כל הנתונים כאילו הוא עכשיו רץ.

מתי להשתמש בסביבה שתומכת במולטי פרוסס?

- כאשר יש הרבה בעיות שצריך לטפל בהן במקביל
- סלקט לא מתאים.
- פרוססים נוצרים באופן לא סדיר / תדיר.
- מספר נמוך של פרוססים סה"כ.
- רוצים הגנה

מתי לא להשתמש בפורססים?

- מתי שאנחנו יכולים לא להשתמש
- כשהרבה אינפו' עוברת
- כשצריכים ביצועים גבוהים (קונטקסט סוויץ' הוא יקר)
- כשת'רד מספיק טוב.

ת'רדים

- יש להם זכרון משותף
- ת'רדים הם מיני-פרוססים – הם משתפים את ה-heap שלהם, את הסביבה והמשתנים הגלובליים.
- לכל ת'רד יש stack משלו. (אני יכול לשים מצביע לסטק של תרד אחר ובכך לגשת לסטק שלו, אבל זה לא בטוח – כי כשהוא ימות יהיה לי מצביע לזיכרון לא מאולקץ ***)
-

קריטיקל סקשן – אזור קריטי

1. מקום בזיכרון שלשתי משימות (שם כללי) יש גישה אליו: קורה עם משימות בעלי זיכרון משותף כמו ת'רדים.
2. הסיכון – אם שניהם מנסים לכתוב בזמן שאחד קורא, או מנסים לכתוב במקביל – יוצא מחורבש.

איך מטפלים באזור קריטי

1. להמנע מזה. (הערכות מעריכות כי ב- 80% ניתן להפטר מאיזור קריטי)
2. באמצעות locking / mutex:
   a. Mutex – אני תופס את המקום הזה בזכרון, עד שאני משחרר אותו.
      i. אומר שתפסתי ראשון, זה שלי, עד שאני לא משחרר את זה. מי שאמר ראשון קיבל (ילדים שנכנסים לאוטו)
      ii. לא באמת נועל את הזיכרון – אלא זו המלצה שמצפים מת'רדים לקיים (כלומר מצפים מהת'רדים לבדוק אם שמו מוטקס קודם) .
      iii. נעלתי מוטקס. מה קורה כשאני נועל אותו שוב או מישהו אחר נועל אותו גם?
         1. תלוי באימפלימנטציה. יש כאלו שאפשר לנעול את המוטקס 800 פעם, וברגע שאני משחרר אותו אני צריך לשחרר אותו 800 פעם כדי שהוא ישתחרר. ב-SUN - מספיק שחרור אחד. בעולם של לינוקס, כל נעילה וכל שחרור נספר. רק כשמספר הנעילות הוא 0, אחר יכול להיכנס.

b. Cond – "אני שני" - כמו מוטקס אבל הפוך - אומר שאני מוכן, עכשיו אני מחכה שיגידו לי שמה שאני אמור לטפל בו מוכן בשבילי. אומר: אני מחכה לך שתגיד לי מתי אני יכול להתשמש בזה.***

3. Risk memory overrun – ואז צריך להוציא אותך החוצה ולירות בך.

API של ת'רדים –

אנחנו נעבוד עם posix 95. יש יותר אחרים.

פונקציות של ת'רדים ב-API

- Pthread_create
  1. יוצר ת'רד חדש – מקבל איזו פונ' תהיה ה-MAIN שלו.
  2. אפשר לעשות wait גם ל- thread שלך, אבל רק אבא שלו. (כלומר לאמץ אותו).
- Pthread_mutex –
  1. יוצר מוטקס
- Pthread_cond – יוצר קונד.

1. אם במוטקס התחלתי במצב בו אני נועל ואף אחד אחר לא יכול להיכנס - בקונד המצב הוא הפוך. קודם נותן לאחר להיכנס, כשהוא מסיים הוא מעיר אותי ואומר שעכשיו אני יכול להיכנס. כל אחד יכול לשלוח לי קונד.
   - פרודוסר מייצר עבודה, כשהוא מוכן אומר לקונסומר שיתעורר.

קוד לדוגמא: Thread wrapper Class

הראה דוגמא לאיך יוצרים class בשם Cthread שיוצרת לנו ת'רדים. ליצור שכבה על API שיהיה לנו קל לממש. (זו סתם דוגמא להראות שניתן להכין לנו מראש "מייצר תרדים", אנחנו משתמשים ב- C אז זה לא ממש רלוונטי. כמו כן, הוא דאג שהקוד יעבוד גם בוינדווס, מה ששוב לא רלוונטי לנו)

```
class CThread
{
      public:
            CThread(CMutexedSignal* FinishedSignal = NULL);
            virtual ~CThread();
            virtual void * Execute() = 0;
            int CreateThread();
      void WaitFor();
      void Terminate();
      thread_t Thread;
            CMutexedSignal* FinishedSignal;
           bool Started;
     bool Finished;
};

int CThread::CreateThread()
{
#ifdef WIN32
   HANDLE Thread;
   DWORD ID;
   Thread = ::CreateThread(NULL, 0, call_start, (void*)this, 0, &ID);
   this->Thread = Thread;
   return (int)(Thread == NULL);
#else
   return ctf_thread_create(&Thread, call_start, this);
#endif
}
```

```
MAIN:
#ifndef WIN32
   extern "C" {
      static void * call_start(void * This)
#else
         DWORD WINAPI call_start(LPVOID This)
#endif
         {
            if (This) {
               ((CThread *)This)->Started = true;
                     ((CThread *)This)->Execute();
               ((CThread *)This)->Finished = true;
               if ( ((CThread *)This)->FinishedSignal)
                  ((CThread *)This)->FinishedSignal->signal();
            }
            return NULL;
         }
#ifndef WIN32
   }
#endif

Ctf_create_thread
inline int ctf_thread_create(pthread_t *thread,
      void* (*start_routine)(void *), void* arg)
{
#ifdef Solaris_threads
   return thr_create(NULL, (size_t)0, start_routine, arg, 0, thread);
#else // POSIX THREADS
   return pthread_create(thread, NULL, start_routine, arg);
#endif
}
```

למה ת'רדים טובים לי?

- דרך שמאוד קל לתכנון דברים שיעבדו יחד
- יותר קל ל-OS לייצר אותם מאשר פרוססים, בגלל שהכל משותף, לא צריך לשכפל.
- יש לי גישה מכל תרד למשתנים של תרד אחר - אני לא צריך להגדיר פרוטוקול.
  1. אם יש שני פרוססים ואני רוצה לקבל משתנים (נגיד משחק רשת) -רוצה לשאול פרוסס מה יש לו בנק' מסוימת.
- ***ת'רדים זה קול, ת'רדים זה סקסי.***

אילו בעיות נקבל ברגע שאנו כותבים ת'רדים?

- אין הגנת זכרון – כי הזכרון משותף.
- על אחריות המתכנת לפתור את האזור הקריטי – הוא צריך לנהל את כל החלק של נעילה לפני שאני כותב וכו'.. כשעובדים עם פרוססים מערכת ההפעלה לא נותנת שזה יקרה.
- Context switching בין תרדים הוא זול, אבל עדיין לא יעיל כמו פרוסס בודד. עדיין עדיף פרוסס בודד על כמה תרדים.
- בגלל שתרדים זה קול וסקסי, משתמשים בהם יותר מדי. לנסות להעיף אותם בסופו של דבר זו משימה נפוצה בדרך כלל בפרוייקטים.

בטיחות Thread-ים

דוגמא: פונ' (שצריך להרוג את מי שמשתמש בה) ב-C שנקראית strtok – לוקחת סטרינג ומחלקת אותו לטוקנים. מי שמשתמש בזה –באמת- צריך להרוג אותו.

- מה היא עושה? אנו רוצים לקבל את הערכים ולעצור כל פסיק לדוגמא. זו פונ' שקיימת ב- standard library. בעצם מוציא כמו מילים או טוקנים כשיש ספרטור בסטרינג - או ספרטור הוא רווח הוא מוציא מילים מהסטרינג.
- למה היא גרועה? שומרת את הנתונים שלה ב-global scope, מחזיקה פוינטרים גלובלים לסטרינג. ואז אם פונ' אחרת שרצה במקביל אלי קוראת לה, היא הורסת את הנתונים שהיא שמרה עבורי. כשאני באה לקחת את הנתונים, הם כבר נדרסו.
- ה- char pointer הגלובלי הוא הקטע הקריטי. לא צריך רק להגן על הקריאה ל-strtok, אלא גם על הקטע הקריטי. צריך להכניס קטע קריטי שאומר - כל זמן שהפונ' מפרססת (מלשון process) קובץ קונפיגורציה, אני צריכה למנוע מכל אחד אחר מלהשתמש בפונקציה. אני אגן על התוכן באמצעות ה-mutex. ברגע שסיימתי אני אשחרר את המוטקס. הבא ינסה להוציא מוטקס, לא יצליח ויחכה. אסיים הרצה ואשחרר את המוטקס. ברגע שאני אעשה זאת, מישהו אחר יתחיל לרוץ, ינעל את המוטקס, יתחיל לפרסס strtok וכו' - ברגע שהוא יסיים נוכל שוב לנעול את המוטקס.
- ועל כן המציאו את strtok_r - מקבל סטרינג, ספרטור ומשתנה לשמור לתוכו את המידע, נפטרנו מקטע קריטי. 80% מהקטעים הקריטיים - ניתן להיפטר מהם. ההסבר לגבי strtok_r:
  1. ל-strtok_r אנו שולחים עוד פרמטר. זה בעצם מערך של מחרוזות בו הוא ישמור את התוכן. יצר מקום ששייך רק לו.

```
#include <string.h>
   char *
   strtok(char *str, const char *sep);
   char *
   strtok_r(char *str, const char *sep, char **last);
```

- Ctime() עושה את אותה השטות, על כן המציאו את ctime_r(), localtime_r() וכו' (פונ' שיש להן קו תחתון r, להסיק שהן כאלו).

איך מקמפלים קוד של multi-thread-ים?

צריך להגיד לקומפיילר להוסיף את הספריה שיודעת לעבוד עם ת'רדים:
את זה עושים עם הדגל:
-lpthread

שימושים רעים נפוצים בת'רדים

- ליצור ת'רד לכל התחברות חדשה בשרת, שכל אחד יעשה משהו קטן אחר.

שיעור 4

מצגת *Processess: general description*

השיעור נלמד עקרונות תיאורטיים בנושאי פרוססים. דומים גם בלינוקס וגם בווינדואס.
תוכניות אמיתיות בנויות מהרבה תהליכים –תהליכים נקראים heavy-weight, עבורם ה-context switch הוא יקר. ת'רדים נקראים light-weight, עבורם context switch הוא זול.

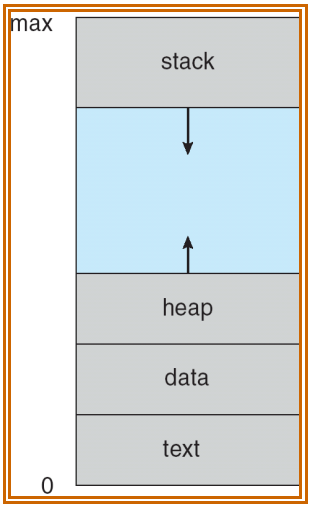
ב-CS (context switch) אין לפרוססים שליטה מתי ירוצו, וה- CS שומר את כל האינפו' שלהם כדי שפרוסס ימשיך לרוץ כאילו לא הפסיק.
המציאו את ת'רדים על מנת לעשות CS יותר זול – לכן יותר מסובך לתכנת ת'רד מאשר תהליך.
(CS לת'רדים לא צריך להחליף את כל הזיכרון, לעומת זאת לפרוסס כן צריך – STACK אחר וכו')

נסמלץ: תוכנה רצה, המתזמן רוצה להעיף אותה ולשים מישהו אחרת במקומה.
אילו נתונים שלה הוא שומר? (נתונים כללים של פרוסס)

- באיזה PC (PROGRAM COUNTER) היא נמצאת.
- באיזה מצב נמצא הפרוסס (ראה בהמשך, שקף 13)
- הרגיסטרים שלה (בהמשך הסיכום)
- הקבצים הפתוחים
- המחסנית שלה ~3
- אזור הנתונים שלה – נתונים קבועים מראש שהתוכנה שומרת, כמו הסטרינגים שלה. נגיד: do you want to save changes? זהו סטרינג קבוע של וורד, שנשמר באיזור הנתונים (data segment).

למינוח: ספרים לעתים במונח job, הכוונה היא לפרוסס.
כשמתכננים CS, השאיפה היא שיהיה שיתוף, כדי שאני לא אצטרך להחליף הרבה. בגלל זה ת'רדים זה טוב, כי הם משתפים זיכרון.

<u>מבנה הזיכרון:</u>



הפוך גוטה הפוך (לא ככה בכלל למעשה, זה רק דוגמא סכמטית. זה קצת בסדר
** הסטאק וההיפ לא יפגשו, הם הולכים לכיוונים שונים ויכולים לגדול כרצונם וכאוות נפשם.

הטקסט הוא הקוד של התוכנית, וכל השאר דיברנו עליו כבר.

<u>שאלה בכיתה</u>: האם בשתי ריצות שונות של אותה תוכנית, נקבל את ה-break באותו מקום? (כלומר ה-break של ה-CS, כלומר כשה-CS עוצר אותנו, נותן למישהו אחר לרוץ). האם בשתי הריצות ה-CS יעצור אותנו באותה הנק'?
התשובה היא שלא – זה תלוי בעומס של המערכת (load). אם אני רץ לבד הוא לא יעצור אותי הרי..

<u>שאלה בכיתה</u>: האם ההחלפה של ה-CS היא רנדומלית? **תשובה**: אין רנדומליות במתזמן, יש מדיניות (בדר"כ היא סוציאליסטית). מבחינתי כמתכנת זה נראה לי רנדומלי כי אני לא יודע את מצב המערכת.

אילו רגיסטרים יש ב-CPU? (ה-CS צריך לשמור את כולם, ראה לעיל) (לא במצגת).

- System registers – יש 16 כאלו.
- PSW – בגודל 24 ביט, כל ביט אומר משהו על התוכנית.
- PC – זוכר באיזו שורה אנו נמצאים.
- SP – ה-stack pointer, שאומר איפה תחתית המחסנית
- General purpose registers – אוגרים לשימוש של היוזר. התוכנית שלי ב-C חייבת לזכור ב-PC את השורה שלה. לעומת זאת ה-general purpose הם אוגרים לתכנות.
- רגיסטרים נוספים ב-CPU.

<u>איך מעבירים קוד C לאסמבלי</u>? (את זה הקומפיילר עושה)

```
c=a+b ;

LR R2 a          (Load Register)
LR R3 b
Add I2 I3  (Adds and save it in I2)
ST I2 c          (Store)
```

(בסוף מחזיר את L2 ל-C).

איכות של קומפיילר תלויה באיך הוא מקצה את הרגיסטרים, האם הוא עושה את זה יעיל.

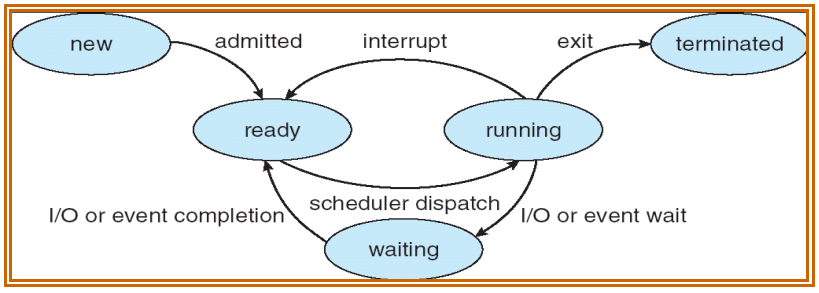
<u>סביבה של פרוסס (שקף 8)</u>
חלק מהסביבה זה דברים שצריך לשמור כגון: האם פתחתי קובץ? האם יצרתי פרוסס?
4~

<u>מצבי פרוססים – process state (שקף 9)</u>

- New – כשחדש, המצב ההתחלתי שלו
- Running – הפק' שלו מורצות
- Waiting – הוא מחכה לאיזשהו מאורע שיקרה (או שהמתזמן יתן לו זמן ריצה, או לקלט מהמשתמש וכו')
- Ready – הוא מוכן לריצה – מצב שהפרוסס מוכן לזינוק. הוא עשה upload של כל ה-info (את זה ה-CS עשה), ומשגרים אותו למשאבים – נותנים לו CPU, זיכרון וכו'.
- Terminated – סופו של כל פרוסס.

פרוסס מחכה רק בתור אחד (CPU, או IO, לחכות לסיגנל וכו'), התור ready הוא עבור פרוססים שמחכים ל- CPU.

גרף שמראה איך הוא עובר ממצב למצב: שקף 10



עוד דיאגרמות בשקפים הבאים.
צריך לזכור לחכות להם (לתפוס את ערך ההחזרה שלהם) את כל התהליכים שיצרת.
אם אני כותב כמה תהליכים, לא מובטח לי שאחד מהם ירוץ לפני השני. אם זה מה שאני רוצה, **אני** צריך לוודא את זה.
איך מוודאים? אחד יעשה wait לשני עד שהוא ימות ואז הוא יפעל וכו'. אחרת ה-CS יתן מה שבא לו.

<u>יש משאבים שהם preempt, ויש כאלו שלא</u>.
לדוגמא – קיבלתי CPU או זכרון או OS. היא יכולה לעשות לי CS מתי שהיא רוצה.
אם קיבלתי מדפסת עם זאת, ה-OS לא יכולה לעשות CS, חייבת לחכות שאני אסיים להדפיס ואז לתת למישהו אחר.

- Preempt – משאבים שניתן לקחת לך
- לא preempt – משאבים שאי אפשר לקחת לך

<u>זיכרון וירטואלי</u>
למחשב מסוים יש גודל זיכרון נתון. אבל אנו לפעמים נרצה שיהיה לנו יותר זכרון. פתרון שאפשר לעשות זה לקחת חלק מההרד דיסק ולהשתמש בו כזיכרון. זה יותר איטי אמנם.

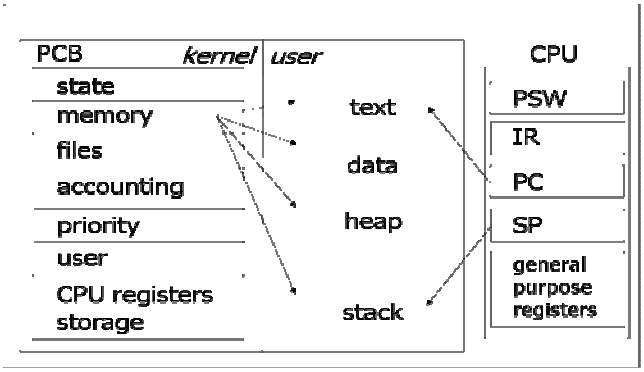
- Swapping in – פעולה שמעבירה משהו מהזיכרון ל-swap, קרי האזור בהארד דיסק שאנו אומרים שהוא עכשיו חלק מהזיכרון
- Swapping out – ההפך.

OS יכולה לעשות אותן מתי שהיא רוצה. כי נגיד יש הרבה פרוססים שרצים, אז אין באמת מקום לכולם בזיכרון. אלו שכרגע לא רצים הם נשמרים ב-SWAP. כאשר מערכת ההפעלה תרצה להריץ אותם, היא תוציא אותם מה-SWAP לתוך הזיכרון האמיתי. גם פעולה זו של הכנסה והוצאה היא מאוד יקרה.

אי אפשר לעשות swapping out מבלי לעשות CS. אי אפשר להעביר תוכנה ל-SWAP, אלא אני אעשה לה CS ואת המידע שלה אני אשמור ב-SWAP. כלומר אנו לעולם לא מריצים תוכנות מה-SWAP.

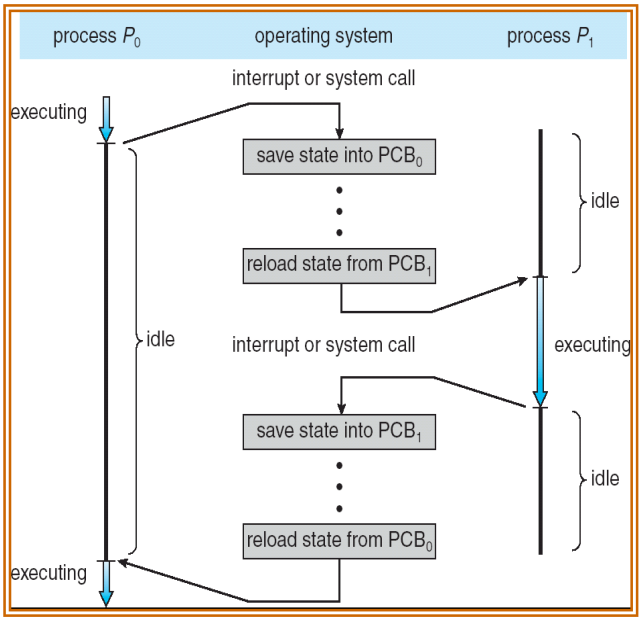
גם כשעובדים מול ה-SWAP, יש את כל המשחק של miss ו-hit (כמו עם הקש). כאשר יש הרבה miss-ים זה יקר.

<u>process control block – PCB</u>
מה שצריך לשמור עבור הפרוסס – גוש נתונים, זה מה שה-CS צריך לשמור. המצב, זיכרון וכו'.
דוגמא ל-PCB:



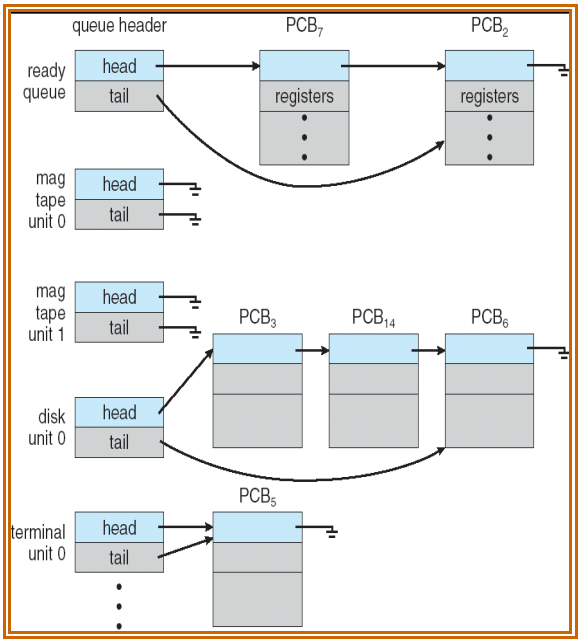
<u>דוגמא לריצה של שתי תוכניות במקביל</u>: (שקף 17)

חץ כחול – התוכנית רצה. קו שחור – תוכנית לא רצה. בהתחלה התוכנית רצה, קו זמן מלמעלה למטה. נמאס לנו להריץ אותה, אנו עושים לה CS, שומרים את ה-PCB שלה, טוענים את ה-PCB של p1, מריצים אותו, ואז הפוך.



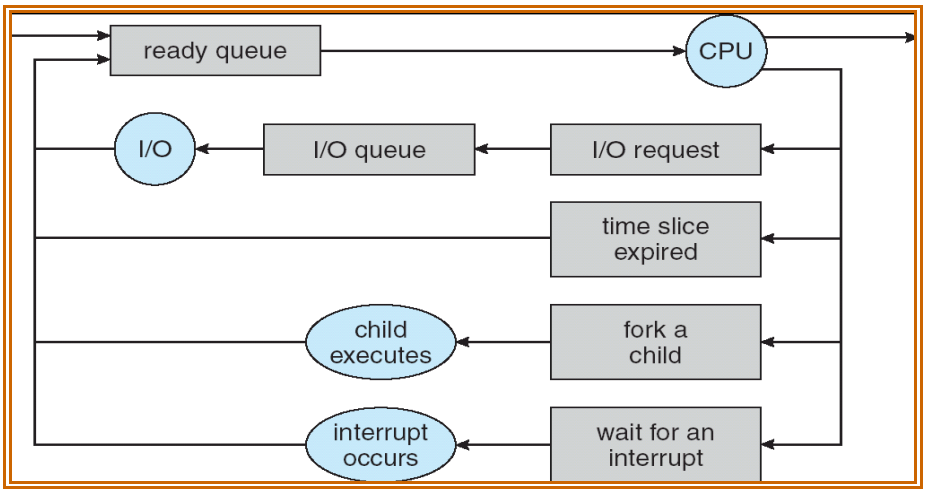
תורת התורים – Process Scheduling Queues

יש ב-OS הרבה תורים – התור של כל אלו שמחכים לרוץ, תור של אלו שמחכים לקלט וכו'.
דוגמא לתורים שממתינים ל-I/O.
בעבר (סביבות שנות ה- 70), ספרים בתחום מערכות הפעלה בעיקר התעסקו בתורת התורים.



בדוגמא יש כמה שמחכים בתור ל-disk, אחד מחכה בתור ל-terminal, אף אחד לא מחכה ל-tape.

התנהלות ה-scheduler:

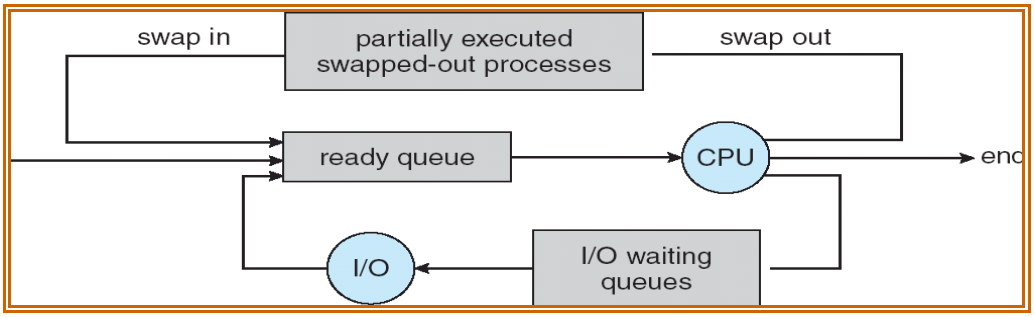


Schedulers = מתזמן

שני סוגים:

- מתזמן לטווח ארוך – מחליט מי הולך להיכנס ל-ready. נקרא גם מתזמן הפרוססים. (מתזמן גו'בים). בסדר שיהיה קצת יותר איטי. (לדוגמא כשאני עושה סלקט, אני מחכה ל- IO ולא נמצא בתור של ready. אך ברגע שסלקט יחזור, כלומר שיש IO, אז המתזמן הזה יכניס אותי לתור ה- ready, ועכשיו אני בידי המתזמן לטווח הקצר שיחליט מתי אני אקבל CPU).
- מתזמן לטווח קצר – מתזמן על ה-CPU, מי יקבל את ה-CPU. מתזמן מתוך האלו של ה-ready מי מקבל עכשיו?. לו קוראים הרבה.

ציור שמשלב גם את התהליך של: swapping in, swapping out:



פרוסיים ניתן לתאר כ:

- I/o bound processes – מתעסק יותר בקבצים (או דברים אחרים שהם לא חישובים, כמו תקשורת) מאשר בחישובים.
- CPU-bound processes – משקיעים יותר זמן בחישובים.

Context Switch

על מנת לייעל את ה- CS, היום יש CS גם בחומרה. פתרון נוסף הוא השימוש בתרדים

יצירת פרוססים

בשקף 25 – 29 – יש הסברים על יצירת תהליכים (קוד של FORK ,wait וכו')

שקף 30 - Process Termination. בנים שנשארים יתומים ומנסים לתקשר עם ההורים שלהם, טוחנים את ה-CPU.

שיטות לתקשורת בין פרוססים (**בעית ספק לקוח**) : מספר דרכים:

- ע"י shared memory – מתאמים על מקום בזיכרון, שדרכו הם ישתפו מידע – אחד יכתוב שם, השני יקרא.
  1. יודעים שה-OS לא תתן לעשות את זה בין פרוססים (מגנה על הזיכרון), אפשר לעשות בין ת'רדים.
- Message passing – פשוט פותחים קו תקשורת (UDP, TCP וכו').

בגלל שאנו לא יודעים מתי כל תוכנית תרוץ, מתי היא תיעצר ותמשיך אחרת, אבל אנו כן רוצים שתוכניות יתקשרו ביניהן, כן צריך להכניס סדר בבלגאן. כל התקשורת בין פרוססים נכנס הרעיון של להכניס סדר בבלגאן אבל לא יותר מדי.

סדר בבגלאן במובן של – כשאני רוצה לשלוח משהו ליישום אחר, אני קודם כל צריך לשלוח, ואז לקבל. אם אין תיאום זה בלגאן. המטרה שלי היא בלי קשר ל-CS, להשיג תיאום.

כי בסופו של דבר הכל סדרתי, המחשבה היא מקבילית וצריך להכניס סדר כשהם מנסים לתקשר ביניהם.

סיכום – דרכי התקשורת בין מעבדים, כאשר יש מצב של ריבוי מעבדים, זה גם אותו דבר – גם:

- או גוש זיכרון שמשותף למעבדים שיתקשרו דרכו
  1. לא מומלץ – כי יש הרבה מעבדים וזה יכול ליצור בעיה של לחץ על התעבורה – מי כותב, מי מחכה וכו'. כל בעיות סינכרון אלו עשויות להוות צוואר בקבוק.
- הפתרון שבנו – switching – במקום שיהיו מעבדים שיהיו מחוברים לזיכרון אחד, כל זוג ביניהם מחובר.
  1. בדיוק כמו ההקבלה לפרוססים.

(**סטייה מהנושא**:

מה שטוב בקרנל של לינוקס זה שהוא **קומפקטי**. כלומר, החליטו מה נכנס וזהו – יותר לא נכנס. היתרונות בזה שהוא מאוד פורטבילי, קל להעבירו ממקום למקום, וגם קל לתחזק אותו. ת'רדים לא נכנסו לקרנל של לינוקס, זו ספריה שצריך לקמפל איתה.)

<u>המשך בעית ספק לקוח</u> (שקף 32)

יש שני סוגי **<u>באפרים</u>** – אחד סופי (בו יכול להיות בעית OverFlow, כלומר אני לוקח הרבה והלקוח עוד לא לקח את המידע), ואין סופי.

בשקף 33 יש ינסיון לפתור בעית ספק לקוח באמצעות shared memory.

יש באפר משותף, שניהם יכולים לקרוא ולכתוב לו.

בהתחלה הוא מגדיר את in, out שניהם ל-0. הבאפר הוא קטע הזכרון היחיד שמשותף להם, רק דרכו הם יכולים להעביר מידע.

- **Shared data**

```
#define BUFFER_SIZE 10
Typedef struct {
  . . .
} item;

Item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```

- **Solution is correct, but can only use BUFFER_SIZE-1 elements**

<u>פונ' Insert()</u>

באמצעותה הספק כותב לבאפר, וכך היא נראית:כמובן שיש בעיה שאם הוא מכניס יותר מדי (נגיד כאן יותר מ-10), והשני לא עושה remove, אז אין לו מקום לכתוב אליו – זו הבעיה של באפר סופי.

הקוד עובד בשיטה של polling וזה לא טוב – זה רק בשביל הדוגמא (לא עובדים ככה במערכת ההפעלה).

```
while (true) {
  /* Produce an item */
  while (((in = (in + 1) % BUFFER SIZE count)  == out)
    ;   /* do nothing -- no free buffers */
   buffer[in] = item;
   in = (in + 1) % BUFFER SIZE;
{
```

<u>פונ' Remove()</u>

בה משתמש הלקוח, באמצעותה הוא מוציא מידע מהבאפר. כל אחד בקצב שלו.

```
while (true) {
  /* Produce an item */
  while (((in = (in + 1) % BUFFER SIZE count)  == out)
    ;   /* do nothing -- no free buffers */
   buffer[in] = item;
   in = (in + 1) % BUFFER SIZE;
{
```

אממה – הפתרון של insert() ו-remove() **<u>לא באמת עובד</u>**. כי אפשר לקבל interrupt בכל רגע, ואנחנו עשויים אפילו לקבל interrupt באמצע השורה אצל השרת: buffer[in]=item;. זה באמצע השורה כי שורת C היא כמה שורות אסמבלי.

וזה בעייתי להפסיק באמצע השורה הזו ספציפית כי הספק העלה את in ב- 1, אבל עדיין לא הכניס את המידע. ואז הלקוח חשב שהוא הכניס.

עוד מקום בעייתי אפשרי – השורה: out = (out + 1).

תרגול 4

מצגת POSIX Threads

ברוב הקוד – אין בדיקות של תקינות לקריאות הפונ' (זה רע, צריך לעשות).

למדנו לתכנת ת'רדים ומוטקסים.

הקוד: יוצר שני ת'רדים ומחכה למותם.
מראה עקרונות איך לשלוח מידע לת'רדים.

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
r
void *print_message_function( void *ptr );

main()
{
    pthread_t thread1, thread2;
    char *message1 = "Thread 1";
    char *message2 = "Thread 2";
    pthread_create( &thread1, NULL, print_message_function, (void*) message1);
    pthread_create( &thread2, NULL, print_message_function, (void*) message2);
    pthread_join( thread1, NULL);
    pthread_join( thread2, NULL);
    exit(0);
}

void *print_message_function( void *ptr )
{
    char *message;
    message = (char *) ptr;
    printf("%s \n", message);
}
```

הסבר על הקוד:
כשרצינו ליצור פרוסס חדש השתמשנו בפורק וזה העתיק בדיוק את התוכנית שלנו והיא המשיכה לרוץ בדיוק מאותה נק'.
בת'רד השיטה היא שונה. כשאנו יורצים ת'רד אנו מגדירים לו פונ' שכתבנו מראש, והיא תהיה הפונ' main של הת'רד.
ראשית מגדירים את המשתנים שלנו (שני תרדים ושני מחרוזות)
הפונ' שאנו רוצים שתשתמש כ-main לת'רד היא:

```
void *print_message_function( void *ptr )
{
    char *message;
    message = (char *) ptr;
    printf("%s \n", message);
}
```

כל פונ' שהיא פונ' מיין של ת'רד, צריכה לקבל מצביע ל-VOID, ומחזירה מצביע ל-VOID. כלומר אני כותב את הפונ' והיא מיועדת להיות main של ת'רד.
איך יוצרים?
באמצעות הפונ': pthread_create.
אנו קוראים לה פעמיים, כי יוצרים שני ת'רדים.

```
SYNOPSIS
    #include <pthread.h>
        int pthread_create(pthread_t * thread,
            const pthread_attr_t * attr,
            void * (*start_routine)(void *),
            void *arg);
```

היא מקבלת דבר ראשון את האובייקט thread, דרכו אנו אקבל מידע, אשלוט של התרד ועוד. הדבר השני – דגלים. אצלנו יהיה null כמעט תמיד. הדבר השלישי – הפונ' שתשמש כפונ' מיין שלו. הדבר הרביעי – הארגומנט שאנו רוצים לשלוח לה. הוא חייב להיות תמיד מסוג מצביע ל-VOID (בשביל זה צריך לעשות casting).
בקוד הוא מנסה להראות שקודם יצרנו ראשון וקודם שני. על כן אנו מצפים שקודם יודפס thread1, ואז thread2. אבל – לא! אנו לא יודעים מי ירוץ ראשון.
אחרי שיצרנו את הת'רדים אנו רוצים לחכות למותם. איך אנו מחכים? עם הפונ' pthread_join. שהיא בדיוק כמו wait עבור פרוססים. זוהי פונקציה חוסמת, blocking, כמו wait.

איך מקמפלים עם ת'רדים? מוסיפים ל-GCC את הספריה lpthread–.

Pthread EXIT

איך ת'רד מסיים? בגלל שהוא פונ' הוא יכול לעשות פשוט return, אבל מעדיפים בצורה מסודרת לצאת באמצעות:
void pthread_exit(void *retval);.
דרכה ניתן להחזיר ערך לאבא שלה. (כמו כן, זה מאפשר לצאת מהת'רד דרך פונקציה אחרת שהת'רד קרא לה)

איך מקבלים חזרה ערך מהת'רד? (לא חובה, כי אין דבר כזה זומבי ת'רד – הוא ימות כשאנחנו נמות ו- init לא צריך לאמץ אותו כמו פרוסס)
Join מחזירה אותו. אבל בצורה כזו:
צריך לשלוח ל-join מצביע לפוינטר ל-VOID, כלומר שבו יישמר ערך החזרה.
int pthread_join(pthread_t thread, void **value_ptr);

שקף 8

בשקף הראשון אנו מגדירים את הפונ' שתהיה ה-main של הת'רד. קודם מגדירים מערך, אותו נעביר לפונ' בהמשך. בזה שאנו מעבירים לפונ' פוינטרל-VOID, אומר בעצם שנוכל להעביר לה כל מידע שאנו רוצים – אפשר להעביר בזה struct שלם, כמו בדוגמא הזו.

```
#define _MULTI_THREADED
#include <pthread.h>
#include <stdio.h>
#include "check.h”
typedef struct
{
        int   value;
 char  string[128];
} thread_parm_t;
void *threadfunc(void *parm)
{
 thread_parm_t *p = (thread_parm_t *)parm;
 printf("%s, parm = %d¥n", p->string, p->value);
 free(p);
 return NULL;
}
```

הפונ' הזו מקבלת את הפרמטר, היא מוציאה ממנו את המידע, ומדפיסה אותו. (הפרמטר הוא struct עם שני נתונים).
כדאי לשים לב לכך שכשמי שיוצר את הת'רד ושולח לו מידע עושה casting למידע להיות void pointer, הת'רד כאשר הוא מקבל את המידע, הוא עושה לו casting חזרה למה שהוא באמת היה (כאילו מסכמים על כך מראש).

<u>המשך הדוגמא:</u>

```
int main(int argc, char **argv)
{
        pthread_t       thread;
  int           rc=0;
        pthread_attr_t    pta;
  phread_parm_t     *parm=NULL;
        printf("Create a thread attributes object¥n");
        rc = pthread_attr_init(&pta);  checkResults("pthread_attr_init()¥n", rc);
        parm = malloc(sizeof(thread_parm_t));
        parm->value = 5;
  strcpy(parm->string, "Inside secondary thread");
  rc = pthread_create(&thread, NULL, threadfunc, (void *)parm);
        parm = malloc(sizeof(thread_parm_t));
  parm->value = 77;
  strcpy(parm->string, "Inside secondary thread");
rc = pthread_create(&thread, &pta, threadfunc, (void *)parm);
rc = pthread_attr_destroy(&pta);
sleep(5);
printf("Main completed¥n");
return 0;
}
```

המיין רצה לשלוח את אותו גוש מידע לשני ת'רדים שונים. בגלל שאנו שולחים את הגוש כפוינטר אז אותו ת'רד יכול גם לשנות את המידע. וזו בעיה, ועל כן אנו ניצור את אותו גוש מידע פעמיים, יהיה לנו שני malloc-ים. כלומר שלא שניהם יקבלו את אותה פיסת מידע.

<u>דברים שבאמת צריך להרוג מי שעושה אותם</u>:

- אחרי שיצרנו ת'רד ואנחנו רוצים לחכות לו, נשתמש ב-pthread_join. יש אנשים שעושים sleep() כדי לחכות לת'רד שלהם (נגיד sleep(5) מתוך אמונה שהתרד יסתיים אחרי 5 שניות). לא טוב לעשות את זה, יש לחכות באמצעות pthread_join.

<u>שקף 13</u> – אפשר לראות את הדגלים.

- # detached state (joinable? Default: PTHREAD_CREATE_JOINABLE. Other option: PTHREAD_CREATE_DETACHED)
- # scheduling policy (real-time? PTHREAD_INHERIT_SCHED,PTHREAD_EXPLICIT_SCHED,SCHED_OTHER)
- # scheduling parameter
- # inheritsched attribute (Default: PTHREAD_EXPLICIT_SCHED Inherit from parent thread: PTHREAD_INHERIT_SCHED)
- # scope (Kernel threads: PTHREAD_SCOPE_SYSTEM User threads: PTHREAD_SCOPE_PROCESS Pick one or the other not both.)
- # guard size
- # stack address (See unistd.h and bits/posix_opt.h _POSIX_THREAD_ATTR_STACKADDR)
- # stack size (default minimum PTHREAD_STACK_SIZE set in pthread.h),

כמעט תמיד, NULL זה בטוח (כלומר בלי דגלים בכלל) ומומלץ.

מוטקס!

pthread_mutex_t – מבנה חדש, שמייצג את המוטקס.

הפונ' בהן נשתמש: (שקף 16)

```
#include <pthread.h>
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_trylock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

הדוגמא: (שקף 15-14)
יש פונ' שרוצה להגדיל את ה-counter.
Counter במקרה הזה הוא משתנה גלובלי, קרי כל הפונ' יכולות לשנות אותו.
פונ' C רוצה לשנות אותו.
מוטקס זה לא באמת דבר שנועל את הזיכרון, אלא הסכם בין כולם. יש לנו שני משתנים גלובליים – mutex1, counter. הם לא באמת קשורים אחד לשני, אלא הקישור לוגי. C לא יודע לקשר ביניהם. מוטקס הוא סתם אוביקט שניתן לנעול אותו ולפתוח אותו, אנו מסכימים שכשהוא נעול אני לא כותב ל- counter.

pthread_mutex_lock(&mutex1); - נועל את המוטקס.
אחרי הגישה לזיכרון של counter, אנו רוצים לשחרר את המוטקס. איך?
pthread_mutex_unlock(&mutex1);

דף 2:
אנו יוצרים שני ת'רדים. מריצים את שניהם, ועכשיו אנו יודעים שהם לא יחד יכתבו ל- counter.
צריך לזכור ש-mutex lock זה פונקציה בלוקינג. כלומר שנתקע עד שמשתחרר. כלומר כאשר אני עושה mutex lock והמוטקס כבר נעול, אני נכנס למצב של cond.

הפונ' pthread_mutex_trylock
int pthread_mutex_trylock(pthread_mutex_t *mutex);
עושה מוטקס ללא בלוקינג.

צריך לזכור שאם עושים trylock צריך לבדוק את הערך החזרה – כי זה לא בטוח, הוא ימשיך גם אם הוא נעול, בשונה מ-lock.

Cond – למידה עצמית רבותיי.

משהו קטן לגבי קונד: כעושים מוטקס אנחנו חוסמים אחרים עד שאנחנו לא מודיעים אחרת, בקונד זה ההיפך –אני חוסם את עצמי עד שלא מודיעים לי אחרת (סוג של מחכה לסימן ממישהו כדי לבצע משהו)

שיעור 5

מצגת Process synchronization, I

נמשיך את הנושא הכללי – סינכרון בין פרוססים. נראה מס' אלג' לסנכרון.
כל תוכנית רצינית היום בנויה מהרבה פרוססים. מערכת ההפעלה לא יודעת איזה אלג' אני מנסה לבנות, ואין לה בעיה לקחת ולהחליף בין פרוססים שירוצו עכשיו – כל פעם יכולה להחליף ושאחד אחר ירוץ.
אם נרצה סינכרון, סדר לוגי בו אני רוצה שהם ירוצו (אחד מחשב משהון, השני צריך להתחשב בחישוב) – אני צריך להיות אחראי לסינכרון. כאן נכנסים אלג' לסנכרון.

המשימות יסנכרנו את עצמן בשתי דרכים: (הרחבה במצגת)

1. Shared memory – יותר פשוט אבל פחות יעיל
2. Message passing – יותר מסובך, פותחים קו תקשורת, יש פרוטוקול – אבל זה יותר יעיל.

מה אנו רוצים מסינכרון?

1. שזה ירוץ כמו שהתכוונו
2. שמשאבים יהיו כמה שיותר משותפים – שלכל אחד לא יוקצב משאבים שיחודיים לו, כי אז ה-CS הוא יקר.
3. שיהיה קל לשימוש
4. שיהיה מודולרי – שהדברים יהיו scalable – כלומר אם בניתי מנגנון סנכרון ל-10 פרוססים ואז הגדלתי ל-100, שלא יגדל באופן לא פרופורציונלי בעליל.

יש לנו 2 פחדים מסנכרון:

1. אני פרנואיד ועושה הכל מאוד בטוח, ובגלל זה זה גורם לי לאבד מקביליות. דוגמא (קיצונית): אני מריצה תוכנית אחת עד הסוף, ורק אז תוכנית אחרת, ובכך מוודא את הסינכון. זוהי דוגמת קיצון, כמובן שיש מקרי ביניים.
2. כשאנשים לא דואגים יותר מדי, נוצר **דדלוק**. כלומר שמרוב שהייתי מתוחכם, 2 משימות מחכות ל-resource מסוים, ועשיתי את המנגנון כל כך רע, שכל אחד מהם מחכה לשני, ולא קורא כלום – אף אחד לא מאותת לשני "תורך".

לא חכמה לכתוב אלג' לשתי משימות. אבל במס' גדול של משימות שרצות יחד, זה יותר מסובך, ויידרש אלג' – צריך שלא נהיה פרנואידי מדי וכן נאפשר מקביליות, מצד שני שלא ייצר דדלוק.

אף מערכת הפעלה לא מטפלת בדדלוק – גם לא ווינדואס. אם הגעתי למצב של דדלוק, בלינוקס הדרך היחידה לצאת מזה זה להרוג את התוכנית.

יש כל מיני אלג' מתוחכמים לזה, אך מערכות ההפעלה לא משתמשות בהן.

נחזור לבעית ספק-לקוח שלנו: (היום אנו נעסוק רק בחלק המשותף, ולא בשאר התוכנית)
נניח שיש אחד שכל הזמן כותב, השני צריך לקרוא את המידע הזה. הם עושים את זה דרך באפר – אחד כל הזמן כותב לבאפר, השני כל הזמן כותב לו.
הבאפר זה הדבר היחיד שמשותף ביניהם.
כאן כבר צריך לוודא שאין קריאה בזמן כתיבה (קטע קריטי), ויש עוד בעיות שצריך לפתור, ועוד במס' גדול של לקוחות וספקים זה עוד יותר בעייתי.

<u>הפתרון הראשון המוצע (במצגת):</u> שקף 10.

יש לנו את הבאפר המשותף, באפור כתוב התוכן.

במקרה הפשוט, העליון – הלקוח קורא מס' תאים לאחר שהספק כותב.
במקרה הציקלי, התחתון – עם מודולו.
In זה המצביע שמצביע על המקום הפנוי הבא.
Out מצביע על המקום הבא שצריך לקרוא ממנו.
הספק כל הזמן כותב ל-in ומקדם אותו, והלקוח קורא מ-out ומקדם אותו.
הם גם עושים בדיקות.

<u>האלג' של כתיבה / קריאה ל/מבאפר (שקף 13-11)</u>
הסבר:
Counter – מס' האיברים האפורים בבאפר (עם תוכן)

```
#define BUFFER_SIZE 10
typedef struct {
        . . .
} item;
item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
int counter = 0;
```

פונ' הכתיבה של הספק:

```
while (true) {

    /*  produce an item and put in nextProduced  */
        while (count == BUFFER_SIZE)
                ; // do nothing
            buffer [in] = nextProduced;
            in = (in + 1) % BUFFER_SIZE;
            count++;
}
```

יש לשים לב שיקר לעשות את ה-while, כי הוא בשיטה של polling. זה רק לצורך הדוגמא, לא עובדים ככה בפועל.
באותו רגע שהספק כותב לבאפר (שלוש השורות האחרונות בקוד), אסור ללקוחות לגעת במשתנים (buffer,in, counter,out). כי כשאני כותב את הקוד אני מצפה שהוא ירוץ במכה, מבלי שבזמן הזה יקראו.

איך?
נוכל לדוגמא להחליט שעד שהבאפר לא מלא – לא נותנים ללקוחות לקרוא. זו גישה קצת קיצונית.

פונ' הקריאה של הלקוח:

```
while (true)  {
        while (count == 0)
            ; // do nothing
            nextConsumed =  buffer[out];
             out = (out + 1) % BUFFER_SIZE;
          count--;
              /*  consume the item in nextConsumed
    }
```

כל עוד אין מה לקרוא – לא עושה כלום.
ברגע שיש מה לקרוא, הוא קורא אותו, מקדם את out ומוריד את ה-counter.
גם כאן נזכיר שמערכת ההפעלה יכולה לעצור אותנו בכל מקום, אפילו באמצע פקודות (בגלל ששורת קוד של C בנויה מכמה שורות אסמבלי).
אבל – אי אפשר להפסיק אותנו באמצעות פק' אטומיות (כלומר באמצע פק' אסמבלי אחת).

Condition כניסה:
בדוגמא הזו ניתן לראות ש- count חיובי הוא המפתח לקטע הקריטי . אם count אינו חיובי אין כניסה לקטע הקריטי.

הקוד לעיל הינו רעיון ש**לא עובד**. (הסבר עם דוגמא בשקף 14)
למה לא עובד?
נסתכל על איך עובד count מבחינת אסמבלי (רגיסטרים):

- count++ could be implemented as

```
register1 = count
register1 = register1 + 1
count = register1
```

- count-- could be implemented as

```
register2 = count
register2 = register2 - 1
count = register2
```

וכאן הבעיה: (נקראת Race Condition)

```
S0: producer execute register1 = count   {register1 = 5}
S1: producer execute register1 = register1 + 1   {register1 = 6}
S2: consumer execute register2 = count   {register2 = 5}
S3: consumer execute register2 = register2 - 1   {register2 = 4}
S4: producer execute count = register1   {count = 6 }
S5: consumer execute count = register2   {count = 4}
```

S4 היה צריך לבוא לפני S2.
ואז ה-count השתבש, אמור להיות 5.

השאיפה שלנו – להפוך את ++counter לכך שיהיה פק' אטומית **אחת**, וכך מערכת ההפעלה לא תוכל לעצור אותנו באמצע שלה.

בדגומא זו ניתל לראות מאיפה מגיעה המילה RACE – כי לא בטוח מי יגיע קודם לקטע הקריטי, ובכלל לא בטוח למי שהגיע ראשון צריך לעשות את זה.
רוצים למנוע race condition, בשביל שפרוסס שסתם במקרה היה מהיר יותר לא יהרוס לנו. אם הגענו למצב של Race אז האלג' שלנו לא טוב.

על מנת שלא יהיה race condition (כדי שהסנכרון יצליח), **האלג' צריך למלא שלושה תנאים**: (שקף 19).
שלושתם חייבים להתקיים במקביל – אם אחד לא מתקיים לא נוכל להבטיח תוצאה נכונה.

1. **Mutual Exclusion** – אומר שאם פרוסס אחד מתוך ה-50 שקיימים נמצא בקטע הקריטי של עצמו (זה אומר שהוא עבר את ה-condition של הכניסה), אז אף אחד מה-49 האחרים **לא** ימצאו בקטע הקריטי של עצמם. הם ייכנסו רק כאשר הוא ישחרר.
   a. **מתי נוצר דדלוק**?
      i. כשיש בלגאן עם ה-condition כניסה (הסבר למעלה).
      ii. יכול להיות שהאלג' שלי מטבעו דפוק – אחד שחרר אבל השני לא נכנס וכו'.
   b. **נקודות שחשוב לדעת לגבי ME:**
      i. שהקטע הקריטי יהיה קצר, והגישה אליו תהיה מהירה
      ii. שלא יהיה לופ אינסופי.
      iii. לא לחכות למשהו בזמן שאני בקטע הקריטי. לפני כן שהוא ישלח לי, וברגע שיהיה לי את המידע אני אכנס לקטע הקריטי. אחרת אני סתם מעכב את המערכת.
2. **Progress –**
   a. **הקדמה** - אם אין אף משימה שנמצאת בקטע הקריטי, ויש מישהו שרוצה להיכנס אך לא נוצרו התנאים (כי מישהו לא כיבה את condition הכניסה), אז הוא יחכה עד אינסוף. מרוב אדיבות אף אחד לא ייכנס, ומי שירצה לא יוכל.
   b. **עיקרון ה-progress** – אומר שאסור לי לדחות את מי שרוצה להיכנס לאינסוף .
      i. אם שניים רוצים להיכנס – שאחד יצליח. בכל מקרה, שתמיד אחד יצליח להיכנס.
3. **Bounded Waiting** – אם פרוסס אחד הגיש בקשה להיכנס לקטע הקריטי שלו, ובקשתו עדיין לא אושרה – יש לשים חסם על מס' הפעמים שפרוססים אחרים יכולים לגשת לקטע הקריטי שלהם.
   a. נגיד אתה כל הזמן רוצה לעדכן וכל הזמן נכנס לקטע הקריטי ולא משחרר וכל השאר מחכים.

איך נפתור את בעיית הקטע הקריטי? (שקף 24)

1. או בתוכנה
2. או בחומרה – יש מערכות שתומכות בכך
3. או שמערכת ההפעלה תעשה את זה
4. או שיש שפות תכנות כמו AIDA שהן לא פופולאריות, שיש בהן את הדברים האלו built in. אבל שם זה אוטומטי ולא יעיל. למי שיש קטע קריטי, עדיף שיעשה בעצמו ולא יסמוך על הקומפיילר, כי הוא יכול לעשות דברים לא יעילים.

פתרון 1 – האלג' של פטרסון (שקף 25)
תפקידו לדאוג שאם פק' לא אטומית נעשית, שאף אחד לא ייכנס באמצע.
זה עובד טוב מאוד, אבל לא פרקטי. כי זה נכון רק עבור שני פרוססים עם קטע קריטי ביניהם.
שני משתנים:

1. Turn – תור מי להיכנס (פרוסס אחד או פרוסס 2)
2. Flag – מערך בוליאני – מי מהפרוססים מוכן להיכנס.

יש שני פרוססים – אחד i, אחד j.
<u>האלג'</u> <u>עבור Pi, כלומר פרוסס i:</u>

```
while (true) {
        flag[i] = TRUE;
        turn = j;
        while ( flag[j] && turn == j)
                ;
           CRITICAL SECTION
        flag[i] = FALSE;
            REMAINDER SECTION
    }
```

<u>While (TRUE)</u> – תמיד מניחים שירצה להיכנס. עושים בכל הקודים, כדי להציג מצב בו הוא כל הזמן מנסה לגשת. בחיים האמיתיים – אין את ה-while החיצוני.
דבר ראשון הוא מכריז: אני מוכן להיכנס (ה-FLAG שלי הוא TRUE).
אחרי זה הוא מכריז שהתור של J. (כלומר הוא קודם כל אדיב, מוותר לשני).
כל עוד – גם j מוכן להיכנס וזה גם התור של J – אל תעשה כלום, חכה (פק' ריקה).
צריך לזכור ש-Pj מריץ קוד מקביל. בסוף הקטע הקריטי j אומר שהוא כבר לא צריך את הגישה.
Critical Section – כאן יהיה הקוד שמטפל בקטע הקריטי.
אחרת – i נכנס ל-CRITICAL SECTION, ואז יוצא (ומסמל שסיימתי באמצעות flag[i]=FALSE).

גם כאן חשוב לזכור שמערכת ההפעלה יכולה לעצור אותי בכל פקו' (turn = i זו פק' של בערך שבע פק' אסמבלי). ואפילו שכך, הדבר עדיין עובד.
לא עשינו הוכחת נכונות לאלג' הזה, אבל אם קצת נחשוב נוכל להבין למה הוא עובד (גם אם המערכת תעצור אותנו באמצע פקודות).

<u>2 פתרונות שמתבססים על חומרה:</u>

TestAndndSet Instruction - שקף 28

```
boolean TestAndSet (boolean *target)
    {
       boolean rv = *target;
       *target = TRUE;
       return rv:
    }
```

הפק' – משנה את הערך ל-TRUE, אבל מחזירה את הערך המקורי.
בפק' הזו יש שלוש שורות, המון שורות אסמבלי. אבל מה שיפה זה שבמעבדים הפכו את זה לפק' אטומית.
נראה אלג' שמנצל אותה כפק' אטומית שפותרת את הבעיה.

```
  while (true) {
        while ( TestAndSet (&lock ))
              ;  /* do nothing
             //   critical section
        lock = FALSE;
             //     remainder section
    }
```

כשעושים testandset – חוזר אלי הערך והוא ננעל (על TRUE). אם חזר FALSE – המשמעות היא שהוא לא היה נעול ואני נעלתי – והתנאי לא מתקיים, אני יכול להמשיך הלאה, לעבור לאזור הקריטי.
אם חזר אלי TRUE – זה אומר שמישהו שמה, ואז לא משנה שעשיתי SET. אני צריכה לחכות עד שמישהו יעשה lock=FALSE.

SWAP – שקף 30

```
void Swap (boolean *a, boolean *b)
{
      boolean temp = *a;
      *a = *b;
      *b = temp:
}
```

גם לזה בחומרה פיתחו דרך להפוך את שלוש השורות ב-C לפק' אטומית אחת של אסמבלי.

```
while (true)  {
       key = TRUE;
       while ( key == TRUE)
             Swap (&lock, &key );

              //   critical section
        lock = FALSE;
              //     remainder section
}
```

Key = TRUE – מכריז לעצמי שאני רוצה להיכנס.
כל עוד המפתח הוא TRUE, כלומר שאני נועל, אני מחליף בינו לבין המוטקס (lock – משתנה שאומר האם אני יכול לגשת או לא. כאשר TRUE אני לא יכול להיכנס, כבר תפוס).
אם ה-lock היה FALSE, עכשיו החלפתי אותו ל-TRUE, ו-key קיבל את ה-FALSE שלו. ועל כן ה-while מפסיק להתקיים ואני נכנס לקטע הקריטי.
אם ה-lock היה TRUE – אז בעצם ה-SWAP לא עושה כלום, while ממשיך לרוץ עד שמישהו ישחרר את ה-lock.

Semaphore (שקף 32)

שיטה לסנכרון כדי לחסל את ה-while, כדי שלא יהיה polling. נדבר עליה מאוחר יותר.

אלג' Bakery

מתמודד עם n פרוססים.

הרעיון של האלג' –

כל מי שרוצה לוקח מספר (הוא יחיד ומונוטוני עולה. ממש כמו בקופת חולים).
למה צריך מספר ולא פשוט תור? כי יכול להיות שפרוסס מגיע ממערכת אחרת.

בא פרוסס חדש – הוא לוקח את המספר הכי גבוה +1.
באלג' יש בעיה שלפעמים שניים יכולים לקחת את אותו המספר – ואז קובעים מי קודם לפי ה-process ID.

$$(number\ [j],\ j) < (number\ [i], i)$$

משמעות הסימון – קודם משווים לפי ה-number. אם הוא שווה, אז נשווה לפי ה-process ID.

המשתנים המשותפים בין הפרוססים:

1. מערך בגודל n בשם **choosing**, שהוא מערך בוליאני, בו כל פרוסס כאשר הוא בוחר מספר מודיע לכולם שהוא בוחר (הופך choosing[i] ל-True).
2. מערך בגודל n בשם **number**, מכיל int-ים. מכיל את המספרים שהם לקחו בתור.

איך פרוסס בוחר מספר?

$$choosing\ [i] = true\ ;$$
$$number\ [i] = \max(\ number\ [0], number\ [1], \ldots number\ [n-1]) + 1;$$
$$choosing\ [i] = false\ ;$$

מודיע לכולם, ה-number שהוא בוחר הוא המס' הבא בתור. באמצעות הפונקציה max.

האלג' עבור פרוסס Pi:

Process P_i
do {

```
choosing[i] = true;
number[i] = max(number[0], number[1], ...number[n-1]) + 1;
choosing[i] = false;
for(j = 0; j < n; j++) {
    while(choosing[j]);
    while((number[j] != 0) && ((number[j], j) < (number[i], i)));
}
```

```
    critical section
number[i] = 0;
    remainder section
} while(1)
```

דבר ראשון אני בוחר מספר.
אחרי זה – לפני שאני נכנס לקטע הקריטי, אני מוודא שני תנאים, ועד שהם לא מתקיימים אני מחכה:

1. שכולם יסיימו לבחור מספרים, כי יכול להיות שאחד בא לבחור מס' מהתור לפני. עד אז אני מחכה (פק' ריקה)
2. אם פרוסס אחר, j, מחכה בתור (כלומר יש לו מספר שונה מאפס. מס' אפס אומר שהוא לא בתור), והוא לפני (כלומר המספר שלו קטן משלי), אני מחכה. (פק' ריקה). זו ההשוואה לפי המס', ואם אותו מס' אז לפי PID.

<u>תחילת הוכחת הנכונות (המשך בשיעור הבא) – מצגת Bakery Proof.</u>

<u>נכונות</u>:
אם j הוא בקטע הקריטי, אז ה-number שלו שונה מאפס, והוא קטן מיתר המספרים האחרים.
זה המשפט - שאומר שלפי התהליך אם יש מישהו בקטא הקריטי, המספר שלו שונה מאפס, והוא קטן מכל המספרים האחרים. אם נוכיח את זה, ניתן להוכיח את המסקנה שמתקיים ה-mutual exclusion, ה-bounding וה-progress.

ההוכחה:
Bakery algorith (עבר למצגת אחרת)
הוא סימן את האלגו' בקבוצות:
R - ההתחלה
T - כל החלק של ההכנות לקטע הקריטי.
D - בוחרים את המספר
T-D - שם בעצם קובעים אם המספר הזה מאפשר לי להיכנס פנימה
C - קטע קריטי
E - exit (number[i]=0)

יכול להיות מקרה, שבסוף אחרי הקטע הקריטי, צריך לסמן לעולם שעזבנו את הקטע הקריטי. יכול להיות שאם לא נסמן את היציא הנכון, גרמנו לדדלוק - כולם מחכים לסיגנל אשר לא יגיע. ואז אחרים לא ייכנסו.
במקרה הזה - זה לא יקרה. האלגו' כפי שהוא מנוסח כאן, מקיים את שלושת התנאים שדיברנו עליהם.

<u>למה 1</u>
עבור שני פרוססים pi,pj, pi הוא בתוך C (כלומר בתוך הקטע הקריטי), pj נמצא או ב-C או ב-T-D (ההכנות לכניסה לקטע הקריטי)
טענה - אם הוא נמצא באחד משני המקומות האלו - אזי המספר של i קטן מהמס' של j.

<u>הוכחה של למה 1</u>
נסמן ב-s את המקום בציר הזמן שתנאי הלמה מתקיימים:
ש-pi נמצא בקטע הקריטי, ופרוסס pj נמצא או בקטע הקריטי עצמו או בהכנות אליו.
אם pi נמצא בקרע הקריטי, numberi כבר נבחר.אם Number[j] נמצא ב-T-d, המספר שלו נבחר. אם בקטע הקריטי - ברור שמספר נבחר. כך שבשני המקרים numberi ו-numberj קבוע.

אזי קיים זמן לפני S, t1, שבו pi ביצע את choosing[j], שכן pi נמצא בקטע הקריטי. T1 לפני הקטע הקריטי.
אזי קיים זמן נוסף, בין t1 ל-s, נקרא לו t2, שבו הוא בדק ...?
או ש-numberj נבחר לפני t1, או בין t1 לבין S.
נראה מי מהם קיים, ונקבל סתירה, בשיעור הבא.

שיעור 6

מצגת Bakery algorithm

<u>הקדמה:</u>

היום נסיים עם סנכרון ונתחיל ללמוד על תזמון – בצורה נלמד בצורה כללית, ואח"כ מצגת שתתמקד בלינוקס.
לכל אלג' מתוחכם שאנו עושים, חייבים להוכיח את נכונות שלושת התנאים: Progress, Mutual Exclusion, Fairness.

<u>מצגת Bakery Proof</u>

נמשיך עם הוכחת הנכונות של אלג' Bakery.

<u>צריך להוכיח את הלמה הבאה</u>:

For any $i \neq j$, if P_i is in C and P_j is in C $\vee$ (T-D), then

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

<u>הסבר</u>: אם יש שני פרוססים, Pi,Pj, ו-Pi הוא באזור הקריטי של עצמו (C), ו-Pj נמצא או באזור הקריטי או באזור ההכנה לאזור הקריטי T-D, אז המספר של i יותר קטן מהמס' של j.

<u>איך מזה נוכיח ישירות את ה-mutual exclusion</u>?

כי אם Pj נמצא ב-C ולא ב-T-D, ניתן להחליף ביניהם מבחינה סימטרית, ומזה ינבע ש-Pj<Pi, ולא יכול להיות ששני מספרים קטנים ממש אחד מהשני. לכן הוא חייב להיות בקטע T-D.

<u>ההוכחה</u>:

נסמן ב-S את הזמן שבו תנאי הלמה מתקיימים.

לגבי זמן זה אנו יודעים, שהמספר של i כבר נבחר, וגם המס' של j.

קיים זמן t1 שהוא לפני S, שבו Pi בדק האם Pj עכשיו באמצע בחירה. (זה ה-while הראשון ב-T-D). כמו כן, קיים זמן t2, בין t1 לבין s, בו Pi בדק האם הגיע תורו (זה ה- while השני ב-T-D).

יש שתי אפשרויות:

1) או ש-Pj בחר את המספר שלו אחרי t1 ולפני S.
2) או ש-Pj בחר את המס' לפני t1.

מקרה 1):

המקרה הזה אומר שאחרי ש-Pi וידא שכולם סיימו לקחת מספרים והתחיל לחכות לתורו, פתאום Pj הגיע ולקח מספר. לכן ברור ש-Pj לקח מס' יותר גדול, ועל כן בזמן S יתקיים אי השוויון:

(number[j],j)>(number[i],i)מקרה 2):

שני המספרים נלקחו לפני זמן t1, ולפני t2 כמובן גם, ונשארו קבועים עד זמן s.

בזמן t2, Pi מחכה שיגיע התור שלו, והבדיקה להמתנה נכשלת (כלומר הוא לא צריך להמתין), וזה כי אנו יודעים שבזמן s הוא נמצא ב-C (הנחת הלמה).

במקרה השני הוא בוחר את המספר עוד לפני t1.

המס' של j נבחר לפני זמן t2 והוא קבוע, אז החלק הראשון של הבדיקה לא יכל להיכשל (כי i ו-j שונים). ומכאן נובע שבזמן S:

(number[i],i)<(number[j],j)

בעצם הוכחנו כאן רק את ה-mutual exclusion .

אמרנו שצריך להוכיח אבל גם את ה-progress ואת ה-fairness עבור כל אלג' – נשאיר לקורא המתעניין..

<u>חזרה למצגת</u>:

מצגת Process synchronization, II (שקף 6).

<u>עוד פתרונות לבעיה של סינכרוניזציה, ב-HardWare</u>.

פתרונות אלו הן פונ' שהן לכאורה לא אטומיות, אבל במיוחד בשביל דברים כאלו מימשו אותן ב-HardWare על מנת שיהיו אטומיות.

1) Test and Set (משיעור קודם)

a) היא מחזירה את הערך המקורי ומגדירה את lock להיות True. פונ' העזר reset מאפסת את lock (עושה אותו False).

קוד שמשתמש ב-TS:

```
Shared boolean lock, initially FALSE
do {
        while(test-and-set(&lock))
;
                critical section;
        reset(&lock);
                reminder section;
} while(1);
```

הפרוסס מנסה להיכנס לקטע הקריטי, בודק באמצעות TS אם הוא יכול. אם הקטע לא היה נעול, אז TS יחזיר False, וינעל אותו. בסוף הקטע נשחרר אותו עם reset.

אם לעומת זאת הקטע היה נעול, TS לא ישנה כלום, ה-while ימשיך לחכות (busy waiting) עד שיתבצע reset, lock ישתחרר ואז TS יחזיר False.

אם נשים לב, <u>לא קיים כאן Fairness</u> – כי אם כמה מחכים יחד, ומי שמשתמש משחרר, יש לנו race condition.

את כל הקטע של ה-busy waiting נפתור בהמשך.

2) <u>Fetch And Add (FAA)</u>

```
s: shared, a: local
int FAA(int &s, int a)
{
        temp=s;
        s=s+a;
        return temp;
}
```

מקבלת את s, המשתנה המשותף לכולם, ואת a, מספר מקומי. היא מוסיפה את a ל-s, ומחזירה את הערך המקורי של s.

<u>איך אנו מנצלים את זה</u>?

```
Shared: int s, turn;
Initially: s = 0; turn=0;
```

<u>Process P_i code</u>:

```
Entry:
        me = FAA(s,1);
        while(turn < me); // busy wait for my turn
Critical section
Exit:
        FAA(turn,1);
```

S מייצג את המספר הבא שהולכים לשלף, ו-turn מייצג את תור מי. אלו המשותפים.

הפרוסס בא להיכנס. באמצעות FAA הוא לוקח מספר ומגדיל את המונה ב-1 (זה ה-a), ומחכה עד שיגיע תורו.

כל עוד המספר שהתור עומד עליו עכשיו קטן ממני אני מחכה.

כשאני מסיים את הטיפול אני מגדיל את התור ב-1. (כלומר עכשיו התור של הבנאדם הבא).

עומד בכל התנאים – יש fairness וכו'. אבל גם כאן busy waiting (ב-(while (turn < me)).

<u>עתה נטפל בבעיית ה-busy waiting באמצעות ה-semaphore</u>.

יש דרך שנקראת monitor, לא נלמד אותה.

<u>Semaphore:</u>

הקדמה:

לכל משאב שהוא משותף נרצה לעשות Semphore, יהיה אחד כזה פר משאב, ולא פר מערכת.

בדר"כ ה-semaphore יראה כמה אנשים רוצים לגשת למשאב.

אנו משתמשים בכך על מנת שלא יהיה לנו busy waiting. הסמפור הוא בדר"כ פשוט int.

הערך ההתחלתי שלו מייצג כמה אנשים יכולים לגשת למשאב בו"ז. אם הוא מאותחל ל-1, הוא למעשה עוזר לנו לעשות mutual exclusion.

סמפורים מאוד חזקים, משתמשים בהם איפה שצריך סינכרון.

יש שתי פונ' (שהמציא דייקסטרא), ()V ו-()P.

$$P(S)\{ \quad while(S \le 0); \quad S--; \quad \}$$

$$V(S)\{ \quad S++; \quad \}$$

הפונ' V היא מוסיפה ל-S אחד, P מחכה עד ש-S יהיה חיובי, ואז היא מפחיתה ממנו אחד.

<u>חזרה למצגת:</u>

*Process synchronization, I (*שקף 32*)*

מעתה נקרא ל-P, wait, ול-V נקרא signal.

<u>איך עושים את הסנכרון הזה</u>?

```
Semaphore S;   // initialized to 1
wait (S);
        Critical Section
   signal (S);
```

בעצם יצרנו סמפור ואתחלנו אותו ל-1, והוא מייצג לנו מוטקס.

בהתחלה הוא עושה wait(s), תנאי הכניסה שלו לקטע הקריטי, מחכה עד ש-S חיובי. אם אף אחד עוד לא נכנס, אז ה-int חיובי, כלומר 1, wait מוריד אותו ל-0, ונכנס לקטע הקריטי. בזמן הזה אף אחד לא יוכל להיכנס, כי הסמפור לא חיובי. כשאני מסיים אני מגדיל אותו חזרה ב-1.

אם זה היה 2, אז שני אנשים היו יכולים להיכנס.

Wait ו-signal הן אטומיות.

<u>סמפור בינארי</u> – לא נדבר עליהם.

לכאורה עכשיו הסמפור במצב הנוכחי שלו לא פותר busy waiting כי יש בתוכו while. אבל הוא לא באמת ממומש כך, אלא כפי שמתואר בשקף 36:

במקום לחכות אני מכניס עצמי לתור של ה-OS, והיא מעירה אותי בזמן המתאים.

Implementation of wait:

```
wait (S){
        value--;
        if (value < 0) {
                add this process to waiting queue
                block();  }
}
```

Implementation of signal:

```
Signal (S){
        value++;
        if (value <= 0) {
                remove a process P from the waiting queue
                wakeup(P);  }
}
```

שימוש לא נכון ב-wait ו-signal, עשויות לגרום ל**דדלוק**, ומערכות הפעלה לא מטפלות בכך.
בשקף 37 ניתן לראות את זה.

P_0	P_1
wait (S);	wait (Q);
wait (Q);	wait (S);
.	.
.	.
.	.
signal (S);	signal (Q);
signal (Q);	signal (S);

מצב של Starvation:
שמנו מישהו ברשימת המתנה והוא נשאר שם לנצח.

נשים לב ש-interrupt לא יכול לקרות באמצע ה-wait, כי זה אטומי, אבל הוא יכול לקרות בין ה-wait-ים.

<u>נחזור לבעיה הראשונית של ספק לקוח</u>:
יש לנו עכשיו שלושה סמפורים. אחד מוטקס שמאותחל לאחד, אחד full שמאותחל ל-0, ואחד empty שמאותחל ל-n.
בשקף 40 אפשר לראות את <u>קוד הספק</u>:

```
while (true)  {
// produce an item
wait (empty);
wait (mutex);
// add the item to the  buffer
signal (mutex);
signal (full);
```

}

הוא דבר ראשון מחכה ל-empty, ואח"כ למוטקס.
אחרי זה אמפטי יהיה שווה n-1. מוטקס יהיה שווה 0, הורדנו אותו באחד.
בסוף הוא משחרר את המוטקס, אבל הוא עושה סיגנל ל-fullף ומעלה אותו באחד.
כלומר באמצעות סמפור עשינו בדיקה האם הבפאר שלנו מלא או לא – את empty אנו רק מורידים ולא מעלים, ואת full אנו רק מעלים ולא מורידים. הלקוח יעשה בדיוק הפוך.

<u>הקוד של הלקוח</u>:

```
while (true) {
        wait (full);
        wait (mutex);
              //  remove an item from  buffer
        signal (mutex);
        signal (empty);

             //  consume the removed item
}
```

הוא עושה אותו דבר אבל הפוך. כלומר אם הספק יכתוב המון עד שכל הבאפר יתמלא, הסמפור של empty יורד ל-0, ולכן הספק לא יכול לכתוב יותר, ורק כשהלקוח יקרה הוא יעלה את empty ויאפשר עוד כתיבה.
וכמובן הפוך עם ה- full - הוא לא יכול לקרוא כל עוד אין תוכן.

עוד בעיה קלאסית:
<u>בעית קוראים כותבים</u>
הפרוטוקול שאנו רוצים יאפשר לכמה שיותר לקרוא ורק אחד יוכל לכתוב בו"ז.
כשאני רוצה לכתוב, אני צריך לחכות שהשאר יסיימו.
יכולים להיות הרבה פרוטוקולים, כמו כשאני בא לכתוב רק הקטע שאני כותב אליו ייחסם, ולא כל הקטע המשותף.

בנוסף אם הוא מחכה צריך לוודא שהוא לא יחכה לנצח (starvation).

<u>יש לנו שלושה דברים משותפים</u>:
שניים שהם סמפורים:
1) Mutex שמאותחל ל-1
2) Wrt שמאותחל ל-1
מספר רגיל:
Readcount שמאותחל ל-0.

יש את השטח המשותף ביניהם.

<u>הקוד של הפרוסס של הכתיבה</u>:

```
 while (true) {
        wait (wrt) ;

           //    writing is performed
        signal (wrt) ;
 }
```

Wrt מסמל את המוטקס שחוסם לכתיבה, וכאן הוא מחכה עד שהוא ישתחרר, וכשהוא מסיים את הכתיבה הוא עוד פעם משחרר. מי שנועל אותו זה הקוד של הקריאה, הוא מסמן מתי הוא יכול להיכנס ומתי לא. אין אפילו שימוש במוטקס כאן, כלומר הוא אפילו לא מוודא שהוא ייכנס לבד, כל הנטל הזה מוטל על הקוראים.

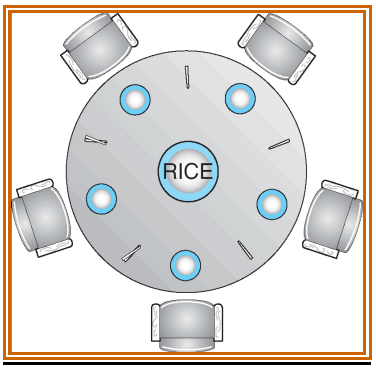
<u>הקוד של הפרוסס של הקריאה</u>:

```
while (true) {
        wait (mutex) ;
        readcount ++ ;
        if (readercount == 1)  wait (wrt) ;
        signal (mutex)

            // reading is performed
        wait (mutex) ;
        readcount  - - ;
        if (redacount  == 0)  signal (wrt) ;
        signal (mutex) ;
    }
```

הקוד ה'אמיתי' – לפני שהקורא משנה את המשתנים המשותפים, הוא מפעיל מוטקס (של 1). הוא מגדיל readcount ב-1, כלומר יש עוד אדם שקורא, והוא בודק.
אם readcount=1, כלומר הוא הראשון שמתחיל לקרוא, אז הוא נועל את wrt. כלומר שעכשיו אם הכותב כותב, הקורא יהיה במצב של המתנה. ולהפך.
אחרי זה הוא קורא, אחרי ששחרר את המוטקס. לקטע של הקריאה מכאן יכולים להגיע הרבה אנשים. ואז הוא צריך לשנות שוב את המשתנים המשותפים, ונועל את הכתיבה אליהם באמצעות mutex, ומוריד את readcount ב01.
הוא בודק: אם RC=0, כלומר הוא האחרון שיוצא, אז הוא משחרר לכתיבה.
הקוד קצת דפוק, כי אם כל הזמן אנשים נכנסים לקרוא, מי שבאמת כותב ירעב למוות, כי אף פעם לא ישחררו אותו לכתיבה.

<u>Dining Philosophers Problem</u>



יש חמישה פולוסופים שיושבים לאכול צהריים יחד, ויש 5 צ'ופסטיקים כמתואר בציור. כדי שפילוסוף יוכל לאכול הוא צריך להרים את שני המקלות, ואז הוא לוקח לזה שלצדו.
הבעיה: אנו לא רוצים ששניים ירימו את אותו מקל יחד.
נרצה לפתור את הבעיה.

```
shared data
        Bowl of rice (data set)
        Semaphore chopstick [5] initialized to 1
```

קוד לדוגמא:

```
While (true)  {
      wait ( chopstick[i] );
            wait ( chopStick[ (i + 1) % 5] );

                  //  eat
            signal ( chopstick[i] );
            signal (chopstick[ (i + 1) % 5] );

           //  think
}
```

האם זה עובד או לא?
לא. כי יכול להיות מצב שכולם ירימו צ'ופסטיק ביד ימין, עשוי להיות דדלוק. כלומר הספקתי להרים ימין, לא הספקתי שמאל, ומערכת ההפעלה קפצה לפילוסוף הבא שעשה אותו דבר בדיוק וכו' – לכולם.

לסיכום – יש הרבה מאוד בעיות עם סמפורים. Tough luck. לא טריוויאלי לעשות אותם.

נושא חדש:

תזמון

מצגת CPU Scheduling

מצגת CPU Scheduling, ch5.
נלמד על תזמון CPU וניהול זכרון.
נצר ידבר יותר על file systems.

<u>מה זה תזמון</u>?
עד עכשיו דיברנו על סינכרון בין פרוססים, כאשר את התזמון עשה ה-scheduler של ה-OS, שיכל לעשות לנו context switch.
תזמון זה המנגנון שבו אנו מתזמנים מי ירוץ עכשיו ב-CPU ולמי נעשה context switch.

<u>מה אנו רוצים להשיג מהמתזמן שלנו כשאנו בונים אותו</u>?

1) שה- CPU ינוצל בצורה מקסימלית, שכל הזמן יהיה busy.
   a) יש לנו הרבה משתמשים, לכל אחד הרבה משימות, ואנו צריכים לחשוב איך עושים משהו שיעבוד במהירות.
   b) יש לנו שני משאבים עיקריים – CPU וזיכרון, נלמד איך מנצלים את שני המשאבים בצורה יעילה
   c) יש תוכנות שהן מאוד מוכוונות CPU (חישובים), ויש תוכנות שמאוד מוכוונות IO.
2) throughput – שכמה שיותר פרוססיס יסיימו את ריצתם בנק' זמן. לסיים כמה שיותר פרוססים. לפעמים אנחנו נרצה את זה (לפעמים המטרה שלנו תהיה אחרת).
3) Turnaround time – זמן הריצה של פרוסס (לכאורה לא מושפע מהמתזמן, אבל כן כי זה תלוי במספר הפרוססים, במיקום הזיכרון שלי [אם יש הרבה פרוססים, יכול להיות שאני נמצא עכשיו ב- SWAP] וכו')
4) שהזמן המתנה ל-CPU יהיה קצר
5) שזמן התגובה של היוזר יהיה מאוד קצר

<u>השאלה הכללית של המתזמן</u>: בעית אופטימיזציה – איך עושים אופטימיזציה בין כל אלו.

לנו כמתזמן חשוב לדעת מה התוכנה דורשת יותר – CPU או זכרון.

שקף 4 – דוגמא לתוכנה טיפוסית שבהתחלה רוצה IO, אח"כ CPU וחוזר חלילה.
את זה היינו רוצים לתזמן נכון, שלא יישב ב-CPU ויחכה ל-IO לדוגמא. סתם בזבוז של זמן CPU.

חשוב לזכור ששני המשאבים, CPU ו-IO, הם לא preempt (אבל אי אפשר לקחת לי באמצע, במובן שאם היא קבעה מדיניות היא לא יכולה להפר אותה)

שקף 6 -
המתזמן מנהל תורים:

- Ready הוא התור הכי מהיר – בדר"כ התזמון שלו נעשה בחומרה.
- Waiting – אלו שמחכים ל-CPU.

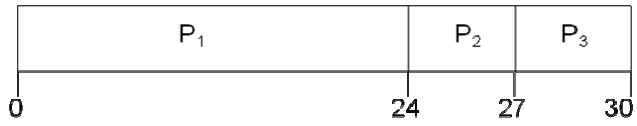
המתזמן יכול להזיז תוכניות מתור לתור.

שקף 7 – **Dispatcher**
כמו מגדל פיקוח בנמל תעופה. הוא נקרא גם short time scheduler.
הוא אחראי על context switch ועוד.

שקף 10 - **דוגמא למתזמן הטריוויאלי**:
הוא מאוד פשוט, אבל לא יעיל (זמן ההמתנה גבוה).
הוא נקרא: first come first serve – FCFS

Process	Burst Time
P_1	24
P_2	3
P_3	3

- Suppose that the processes arrive in the order: P_1 , P_2 , P_3
  The Gantt Chart for the schedule is:



- Waiting time for P_1 = 0; P_2 = 24; P_3 = 27
- Average waiting time: (0 + 24 + 27)/3 = 17

בדוגמא הזו ניתן לראות שזמן ההמתנה הממוצע הוא גבוה – 17 יח' זמן.

נזכור שאם p1 קיבל 24 יח' זמן, אי אפשר לקחת לו את ה-CPU (הכוונה היא שאי אפשר לקחת אותו כדי לתת את ה-CPU לפרוסס אחר. ב-interrupt מערכת ההפעלה כן תיקח לפרוסס את ה-CPU לצרכיה, ואז היא תחזיר לו אותו. אבל זה שקוף מבחינתו).

שקף 11 –
נראה כי אם הפרוססים היו מגיעים בסדר אחר, זמן ההמתנה היה קצר משמעותית.

שקף 12:
המתזמן הבא: Shortest Job First – SJR
נראה שהוא האופטימלי ביותר. הבעיה היא שצריך לנחש כמה CPU ה-job יצטרך.
הוא מאוד תיאורטי, כי רק אם אני אדע כמה כל אחד צריך מראש, אני אוכל לעשות מתזמן אופטמלי.

למתזמן זה יש שתי צורות:

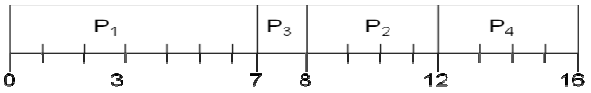
1. Non preemptive – שקף 13
2. Preemptive - שקף 14

Non Preemptive:

בהנחה שאנו יודעים כמה יח' זמן כל פרוסס צריך, מקבלים לפי הסדר

Process	Arrival Time	Burst Time
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4

- SJF (non-preemptive)



- Average waiting time = (0 + 6 + 3 + 7)/4 = 4

הקצר ביותר מקבל, ואנו לא לוקחים לו את זה עד שהוא מסיים.
לדוגמא: אם בא הפרוסס הראשון וצריך 7 יח' זמן, ואחרי 2 יח' זמן מגיע פרוסס אחר שצריך רק 4, אני כבר לא החליף ביניהם, כי בזמן אפס 7 היה הכי קצר.

Preemptive

כל פעם מחשב ביחס לזמן הנוכחי מי הפרוסס שעומד להסתיים הכי מהר ולו הוא ייתן את ה-CPU.
בדוגמא שלנו, פרוסס p2 שמגיע בזמן 2 צריך 4 יח' זמן, ואילו בזמן 2 פרוסס p1 שכבר התחיל לרוץ צריך עוד 2 יח' זמן. ברגע 2 אני אקח את המעבד ואתן אותו ל-p2 כי הוא יסיים אחריו.

בעיות:

1. בשניהם יש את הבעיה של starvation – כי אם כל הזמן מגיעים פרוססים קצרים, פרוסס ארוך לא יקבל לעולם את ה-CPU.
2. לא מעשי, כי אנו לא באמת יודעים כמה זמן כל פרוסס יצטרך.
3. המון context switch והמון בדיקות מסביב.
   a. הנחה ש-CS עולה אפס היא בעייתית.

תרגול 6

<u>שאלות על התרגיל</u>:
מצגת נפרדת.

1. איך תופסים סיגנלים? או עם פונ' signal או עם sigaction, יש ב-beej.
   a. While (wait() - כל עוד שיש פונ' שפרוססים מתו ואנו רוצים לתפוס ילדים.
   b. נעשה signal עם sig child לפונ', והיא תיקרא כל פעם שchild process שלנו מת.
      i. Signal- אינטרפייס ישן פחות גינרי אבל בסדר לקורס הזה.
2. Signal handler
   a. קוד שנקרא בצורה מפתיעה, ואני לא מעוניינים לקרוא לקוד שיכו לקרוא לסיגנלים נוספים. לא ניתן לקרוא ל-sislog עם signal handler. רצוי שכל הפעולות ב-signal handler יהיו פעולות שיחזרו מיד, ולא יחוללו סיגנלים נוספים.
   b. למשל wait על xigchild כשאנו יודעים שיש ילד שמחכה - זה רעיון טוב.
3. איך עושים debugging?
   a. יש debugger טקסטואלי DBB ועוד. צריך לקמפל עם d-.
   b. כשתוכנה שלנו מתרסקת אנו לפעמים רוצים לראות בעצם איפה היא התרסקה. דרך הכי קלה לעשות את זה - לבדוק שהיא תיפול עם core dump. זה image של הזכרון - מה היה בפרוסס כאשר הוא נפל. מקבלים image מדויק של הזכרון.
   c. איך הוא עוזר? נוכל לראות באיזו שורה זה נפל ומה בדיוק היו הערכים ברגע שהפרוסס נפל. זו הדרך הכי פשוטה לעשות debugging.
   d. אם פרוסס תקוע ואנו רוצים לדעת על מה הוא תקוע, ניתן לעשות strace על הפרוסס, נראה את הפונ' האחרונה שיש ב-stack.
   e. נמצא בשקף במצגת.
4. כדי להרוג את עצמנו: kill -9 pid = הורג אחת מהפרוססים.
5. Find:
   a. Grep = פק' של יוניקס, תחפש. *h / *.h Grep etrlala.
   b. כל האינקלודים שנחפש נמצאים ב-usr/include/.
6. על Vi:
   a. מיני פקודות במצגת.
7. מקבלים interrupted system call על select. לקרוא את פרק 10.5 ב- advanced programming in the unix environment.
   a. זה שסלקט נכשל כי אנו מקבלים סיגנל באמצע זו התנהגות צפוי שאמורה לקרות. באמצע סלקט עקרו את מערכת ההפעלה מסלקט ל-signal handler. אנו מצפים שסלקט ייכשל. נכשל עם- interrupted system call.
8. Syslog
   a. לא עבד בתרגיל בית, הסיבה היא security באונ'.
   b. אם נקרא ל-LOG_LOCAL5, ב-openlog
   c. הקבצים נמצאים ב-car/log/messages.
   d. אם נעשה לוגינג לקובץ יקבלו את זה.
   e. בתרגיל 2 - נדרוש סיסלוג.

איך מטפלים בקבצים בניויקס?
שקפי stat.
short cut - Symbolic link, קובץ שמכיל בתוכו path של קובץ אחר, הוא מסומן כלינק. כשמערכת ההפעלה קרואת אותו, במקום לפתוח את הקובץ של ה-short cut, היא קרואת את ה-path, ופותחת את הקובץ שמוביל אליו ה-path.

מה זה struct stat?
מקבלים את ה-device שהקובץ נמצא עליו, את ה-ino שמתאר את הקובץ, את ה-mode של הקריאה (705), מקבלים את ה-user id שהוא הבעל של הקבוץ, את מס' ה-hard links של הקובץ, את ה-group id של הקובץ, סוג הקובץ (נגיד אם symbolic link), את הפעם האחרונה שניגשו לקבוץ. אני קורא קובץ , שזו אמורה להיות פעולה המהירה, אני מיד כותב שנכנסתי לקובץ. יש מערכות שמכל מיני

סיבות, למשל אם זה CD ROM זה read only זה לא נכתב, ויש מערכות שזה לא נכתב מסיבות של ביצועים. אבל בדיפולט כל קריאה כוללת גם כתיבה.

עוד:
הזמן שהסטטוס שונה לאחרונה, גודל של קובץ, מס' בלוקים על הדיסק שקובץ תופס, ו-flag-ים, מידע שנלמד בהמשך.

עוד פעולות שקשורות לקבצים:
Unlink - הדרך שלנו למחק קובץ. אנו מנתקים את ה-deirectory מאחד הקבצים שעל הדסק. ברעג שאין לנו יותר הצבעות לקובץ - הוא ימחק. אם הקובץ שאנחנו יצרנו הוא מוצבע פעם אחת, un link ימחוק אותו.

Lseek - דרך להגיע למקום מסוים בקובץ ולשנו אותו משם לדוגמא (או ל-offset). מקבל שלושה פרמטרים:
1. File descriptor
2. Offset
3. פרמטר שאומר איך אנו רוצים להגיע - האם רוצים לספור ביטים מהתחלה או ממיקום נוכחי.

Create
הדרך שלנו ליצור קובץ בגודל 0.

Dup
הדרך שלנו לשכפל File Descriptor. אם היה socket ונקרא עליו DUP, נקבל שני FD המקושרים לאותו קובץ.

Dup2 - משכפל FD לתוך FD חדש.

Flock
שיטה לנעילה - לא מרחיב עליה.

file control - Fcntl
דרך כללית לגשת לקובץ, דרך עיקרית לנעול קבצים בלינוקס.
יביא דוגמא לכך בהרצאה הבאה.

איך יוצרים symbolic linking?
Symling
Link - דרך ליצור hard link.

<u>ניהול קבצים בזיכרון</u>:
Mmpa - memory map, הדרך שלנו לטעון קובץ ולהכניס את כל הקובץ לזיכרון. בעצם קובץ שלם שהיה על הדיסק עולה לזכרון.
Msync - דרך להשאיר זכרון ממופה אבל גם לשמור את כל השינויים שעשינו לדיסק.

<u>Mmap</u>
- אפשר לנעול את הזכרון עם mprotect (לא בקורס הזה)
- משתמשים ב-sync כדי לשמור, או ב-unmap
- ש"ב - לנסות להעתיק קבצים עם readqwote ועם mmap, מה המיר יותר?

פונ':
Mmap מקבל כתובת לתוכה ייכתב זכרון, מומלץ לתת 0, את האורך של הזכרון שאנו רוצים להעתיק, שני דגלים: אחד להגנה (איך אנו רוצים להגן -זכרון שרק אני יכול לגשת אליו), FD שהוא הקבוץ אותו אני רוצה לקרוא לזכרון, ואופסט על הקבוץ הזה (מאיפה אני רוצה להתחיל להכניס לזכרון).

Unmpa - שחרור זכרון - איזו כתובת אנו רוצים לשחרר ומה האורך.

Msync - כתיבה חזרה לדיסק.

דוגמא:
רוצים להעתיק קובץ.
(שוב יוצא מנק' הנחה שפונ' מצליחות, אנו צריכים לבדוק).
מגדירים stat buffer - שם נקרא אינפו' במאצעות stat.

את ה-dest ניצור אם הוא לא קיים.

נקרא ל- fstat על קובץ ראשון, נקבל באפר שמכיל גם size.
נעשה fseel לקובץ destination ל-size פחות 1, נכתוב byte של זבל.
כשאני מעדכן זיכרון אני יכול לכתוב עד מה שהקציתי, לא יותר.
קורא את ה-source באמצעות mmap. הפרמטר הראשון זו העצה - אם אני יודע איפה הזכרון. לא יודע, כותב 0. מעתיק stat size בתים. מבחינת הגנה נעשה trade ל-shared memory, offset הוא מהתחלה.
אח"כ אני קוראת 0 בתים,..

אני עושה memcopy מ-source ל-dest ל-stat sixze בתים. יוצא מהתוכנית. ברגע זה יוניקס ידאג לי לסינכרוניזציה וישמור את המידע חזרה בקובץ.
אם רוצים להמשיך לרוץ, היינו צריכים לבצע mmap או msync ואז הזיכרון נשמר.
אמור להיות יותר יעיל מ-read ו-write.

שיעור 7

מצגת CPU Scheduling

13-17 – דילגנו

שקף 18 –
יש המון חרטה.
Priority – לא כל הפרוססים עולה המתנה אותו דבר, יש פרוססים שאנו רוצים שיקבלו יותר CPU. אנו לא באמת יודעים כמה זמן כל פרוסס הולך לרוץ. אם היינו יודעים היינו עושים פונקציית עונש אחרת.
איך אנו דואגים שפורסס יקבל יותר CPU? נותנים לו פריוריטי, לפיה יש את ה-dispatcher שיעדיף פרוסס עם פריוריטי גבוה על פריוריטי נמוך. זה לא תמיד חח"ע (ביוניקס מקובל שמי שבפריוריטי נמוך יש לו עונש).
מישהו בפריוריטי 10 חוטף עונש – הזמן הבא שהוא יהיה בעיבוד יהיה עוד הרבה זמן. אם אני אתן לך 100% CPU ויהיו לך באגים, אתה תתקע אותי. אם יש לך פריוריטי נמוך, יכול לבוא מישהו מהשל ולהרוג אותך.
במקומות שיש console, נק' שוורד נתקע לדוגמא, ככה נוכל להרוג אותו.
בגדול כל המערכות שעובדים איתן – לא רץ רק מה שבפריוריטי גבוה.

Round Robin (שקף 19)
אומר שיש חתיכות זמן, כל פרוסס מקבל את החתיכה שלו – ברגע שהיא נגמרת מחליפים ומעבירים לפרוסס הבא. האופן שבו אנחנו מחליטים זה בדר"כ לפי הפריוריטי.
אם יש n פרוססים – כל פרוסס לא מחכה יותר מ-q(n-1) יח' זמן.
המצגת הזו מניחה שכולם באותו פריוריטי.
Q – quantum time.
היום במערכות ניתן לקבוע את q, ויש לקבוע אותו גדול מזמן ה-CS.
אם q מאוד מאוד גדול – יש FIFO. אם זמן עיבוד של משימה זה שעה ו-Q זה שעתיים, זה בעצם FIFO. אם ה-Q מאוד קטן, כל הזמן נעשה CS, אבל עדיין Q חייב להיות יותר גדול מזמן ה-CS.

שקף 20 – דוגמא של איך כל אחד מקבל קצת זמן.
איתי – להכניס ציור.

שקף 21 –
להניח ש-CS לוקח זמן. בעולם מושלם היינו עושים Q של שנייה, CS לא היה עולה לנו, והיינו עושים התקדמות קבועה לכל האורך. אם Q יהיה קצר נבזבז זמן על CS.
מן הסתם כמה ש-Q יותר גדול אנו מנצלים את מעבדי מחשב יותר ולא מבזבזים זמן על CS, אבל זמן תגובה יהיה ארוך יותר.

שקף 23- 24
נדבר על multi level queue –
אומר שיש לנו כמה תורים עם כמה עדיפויות:
יש פריוריטי הכי גבוה שזה סיסטם. יש תהליכים שעולה להם יותר זמן להמתין. בדר"כ אם הוורד לא יגיב מול העיניים נרגיש רע. אבל אם נריץ מטלאב ונחכה חצי שנה, זה פחות משנה מתי הוא רץ.
בעיקרון דברים שרצים ברקע מקבלים פריוריטי יותר נמוך. זה לא מובטח – בווינדואס כשאנו מריצים סרבר, הוא יכול לתת פריוריטי יותר גבוה לפרוססים ברקע – כלומר יש שיקולים משתנים.

שקף 25
תהליכים יכולים לנדוד בין התורים השונים.

נגיד שיש דימון שמקבל פריוריטי נמוך יותר.
איך אנו יודעים שמשהו הוא דימון?
מה שזקן הוא דימון – כי התהליכים של המשתמש הם אינטראקטיביים ומסתיימים, אבל שרת וב יכול לרוץ ימים. ככל שעובר יותר זמן הוא משנה אותו בתורים – בדר"כ יוריד (לפעמים יעלה).

שקף 27 – דוגמאות ל-multilevel queue
להוסיף מתוך מסמך של איתי – לא כל תורים מנוהלים אותו הדבר – יכולים להיות מנוהלים בשיטות.

שקף 29 - Real time –
הגדרות מתוך מצגת

מצגת חדשה –
מצגת *Scheduling Processes in Linux*

בלינוקס יש TQ, בלה בלה.

שקף 3 –
די ברור

שקף 4
שקף 5 –
די ברור. עושה CS.

שקף 8
פרוססים אינטראקטיביים כמו VI –רוצים לתת להם זמן תגובה יותר קצר.
זה לא משהו שמיוחד ללינוקס – נכון בכל מקום.

שקף 9 – לדיומינים באופן עקרוני בדר"כ נרצה לתת פחות זמן.
Batch – דברים שרצים ברקע
Interactive – מה שאנו מגיבים.

שקף 10 – Real time:
אומר שפריוריטי הכי גבוה תמיד יעבוד ראשון. למה זה טוב? אם למשל נרצה קוצב לב, לא נרצה ששום דבר יפריע לו. לעומת זאת, אם אני עובד על ווינדואס וורד קורס, נרצה להיות מסוגלים להרוג אותו.
זה טוב כי אנו לא רוצים שהוא יתקע.

שקף 11
כל ה-system calls.
דביקגורקמע פסרקבט – האם realtimeא ו לא? הדיפולט עדיפות הוא לא ריל טיים.
לעזוב את זה.

שקף הבא –

הטקסט אדיטור אינטראקטיבי, קומפיילר לא – בקומפיילר לא כל כך חשוב לנו הזמן תגובה, באדיטור כן.
מה שצריך להבין זה שבוספו של דבר לכל פרוסס שרץ יש לנו משתנה שהוא פריוריטי – שכל הזמן משתנה.

המזתמזמן משנה אותו לפרוסס. יש לי פריוריטי סטטי שאני מגדיר אותו. המטרה שלך היא לא להיות סוציופט אלא שהמערכת תעבוד.

פריוריטי דינאמי – מבין כל מה שהוא ready, מסתכל מתי קיבלת CPU פעם אחרונה, מעדכן פרויריטי שלך לפי wait צשלך, ונותן לפריוריטי הכי גבוה להתאבד. אם מישהו אחר מגיע לתור, אני מסתכל על ה-ready queue ונותן לו עדיפות. אולי אתה בפרויריטי סטטי מאוד נמוך, אבל בגלל שלא קיבלת הרבה זמן CPU תקבל עדיפות גבוהה ותיכנס ישר.
העונש שלך על זה שנכנסת ל-CPU, כמה הפרויריטי שלך ירד – תלוי בפריוריטי הסטטי שלך.

שקף 19 –
קשקוש.
Swapper זה הפרוסס המחליף את הזיכרון.

שקף 20 –
נורמל זה רגיל.
FIFP זה אומר שפרויריטי הכי גבוה רץ, אלא אם כן מישהו עם פריוריטי יותר גבוה נולד. אפשר להגדיר את זה בין הת'רדים, או למערכת.
שקף 12 – על הת'רדים שלך. (נצר צריך לבדוק אם זה לת'רדים שלי או למערכת).

יש לנו שלושה סוגי פריוריטי:

1. Other – או נורמל, זה אומר שגם פרוססים בפריוירטי נמוך מקבלים CPU רק פחות – כלומר ברגע שקיבלת CPU אנחנו מענישים אותך בהורדת הפריוריטי הדינמי. זה מושפע מהעדיפות הסטטית שלך – ככל שהיא נמוכה יותר, כך תיענש יותר בדר"כ.
2. real time – FIFO מוחלט – הפרחיריטי הכי גובה רץ. לא נפסיק אותו עד שהוא לא יסתיים
3. RR – אומר שאם יש כמה באותה עדיפות, הם ירוצו ב-RR ביניהם, אבל מישהו בעדיפות יותר נמוכה לא ירוץ לפני שכולם יסתיימו. ואז ברגע שהקב' כולה מסיימת, או אחד יוצא – מענישים כמו בנורמל רגיל.

ההענשה – מה שעושים בפריוריטי נורמל. לפי הסטטי שלך.
אפשר להחליט שאם אתה בעדיפות גבוהה יותר תקבל Q יותר גדול.
יש נוסחה שאומרת מה העדיפות הדינמית:

הציורים בשקף 30 –
יש כמה תוכניות, פרוססים – הוא מניח שהכוונה היא שדימונים ולא דימונים.

צריך להבין בעצם מכל החומר הזה

יש שני דברים שאפשר להשפיע בהם:

1. פונ' עונש
2. Quantum time

לסיים יחד עם נצר את השיעור הזה.

שיעור 8

מצגת Memory Management

מה עוד נשאר לנו ללמוד בקורס?

1) Memory management – ניהול זיכרון, על כך נדבר בשיעור.
2) File system
3) קרנל
4) גריד – חישוב מבוזר (הרצאת אורח)
5) מבנה ווינדואס – אם יהיה זמן.

Chapter 8 – main memory

שקף 2 –

הזיכרון – משאב חשוב אך קטן. כל המצגת נדבר בהנחה שיש כמותר בה של משתמשים וכמות עצומה של דרישות, עצומה ביחס לזיכרון.

כלומר רצים המון פרוססים שרוצים זיכרון יותר גדול ממה שיש לנו.

שקף 3 –

אנחנו נראה את ההבדל בין מימושים היום ובעבר – ההבדל העיקרי (רעיון משנות ה-30), שאז התחילו עם הזיכרון הוירטואלי. זה מסובך עד כדי כך שיש יחידה במחשב ליד ה-CPU שנקראת memory management unit (בקיצור MMU) והוא עושה את כל מה שנסביר היום – היחידה היא שממממשת את כל האלגוריתמים.

שקף 4 –

בדר"כ התוכנית יושבת בדיסק, ה-CPU, האוגרים – נטענים באופן זמני. רוב הזמן היא באמת יושבים בדיסק במובן שעושים לה swapping out ו-swapping in.

זה לוקח הרבה cycles לעשות את זה – אחת הדרכים להתמודד היא באמצעות cache. כי מהדיסק זה עובר לזכרון, מהזכרון לקש ומהקש ל-CPU (לרגיסטרים).

איכות קומפיילר נקבעת לפי איכות ההקצאה של האוגרים.

לגבי הקש – מערכת ההפעלה לא דואגת לזה, זה שבארכיטקטורה, וכדאי לנצל את זה שיש קש. איך? שתי דרכים:

1. להימצא באותו אזור בזיכרון למשך זמן – לא לקפוץ מאזורי זיכרון שונים בקריאות אלא לנסות לעשות במרוכז.
2. לא לעשות הקצאות של מערכים ענקיים כי זה לא ייכנס לקש.

כשאני כותב תוכנית, אז אני משתמש בזכרון. אבל כשמריצים אותי, כל פעם הזכרון יוקצה לי במקום אחר. אז כשאני מתחיל את התוכנית מערכת ההפעלה מספרת לי מאיפה הזיכרון שלי מתחיל. אני כמתכנת מניח שהזיכרון שלי מתחיל ממקום 0. כשאני מתחיל להריץ את התוכנית, מערכת ההפעלה מספרת לי כל פעם מאיפה אני מתחיל, ואני מוסיף את זה לכתובת האבסולוטית שלי. למעשה הקומפיילר עושה את זה.

איך הוא מספר לי?

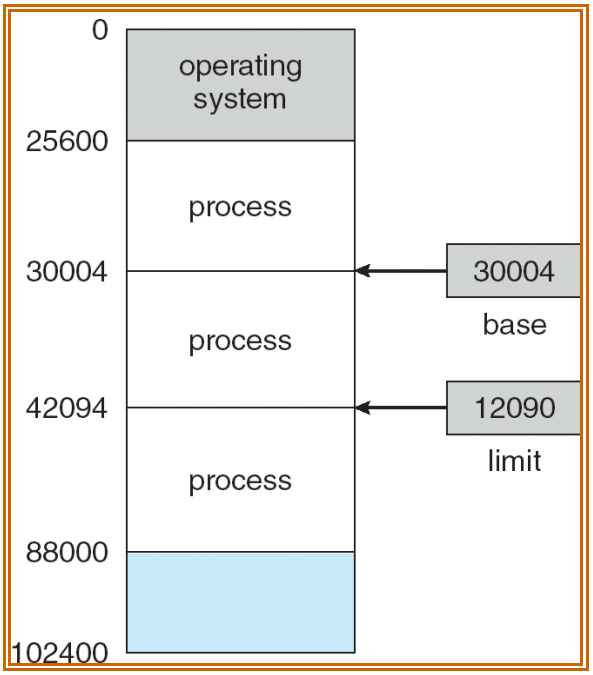
הפק' הראשונה בתוכנית שלי באסמבלי אותה שם הקומפיילר בשבילי:

הפק' הראשונה היא load register עם כוכבית (זה הסינטקס).

```
Load Register R1 , *
```

במקרה זה R1 נבחר להיות ה-base register. (זה שמו של האוגר שמכיל את הכתובת שיש להוסיף לאבסולוטית). אחרי שקבענו את רגיסטר הבסיס כדבר ראשון לא נוגעים בו יותר.
כך, כל פעם שאני רוצה לגשת לזיכרון, נניח למקום 7, אני אגש לזיכרון במקום r1+7.

דוגמא:בשקף 5 נראה את הפרוססים שנמצאים במערכת והרגיסטרים בסיס שלהם.



כך זה נעשה בעבר ולא נעשה היום.
היום פרוסס נשבר לדפים בגודל 1 עד 4 K, והם לאו דווקא יושבים יחד. הם יכולים להיות מפוזרים ויש טבלאות שמסדרות אותם. (אותן טבלאות חייבות להיות מהירות, ועל כן הן ממומשות בחומרה ב-MMU (memory management unit).

שקף 6 –

1. קמפול:
   a) כשאני מקמפל אין לי מושג מה יהיה מצב הזיכרון בהתחלה, ועל כן אני אתחיל ברגיסטר בסיס 0 (אני מניח שהוא אפס ואח"כ הוא ימלא אחרת).
   b) נוסף על כך, אני משתמש בהרבה ספריות שגם הן נמצאות בכל מיני מקומות שאין לי מושג לגביהן.
2. ה-load time, זמן טעינה (כשאני בוחר להריץ את התוכנה, טעינה לזכרון)
   a) עדיין הכתובת שלי היא אבסולוטית, כלומר 0. ורק בזמן ה.. (לקרוא שורה הבאה)
3. Execute –
   a) כאן כבר יודעים כמה פרוססים רצים, איפה הם נמצאים וכו'.
   b) אני מקבל את הרגיסטר בסיס שלי.

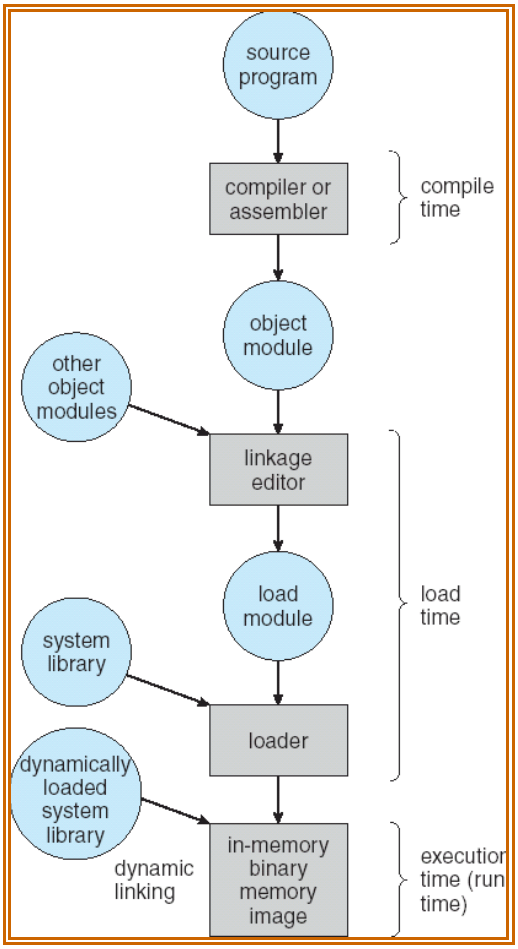
כשאני מריץ את התוכנית אז היא נטענת לזיכרון ונשברת לדפים (לכל דף יש את רגיסטר הבסיס שלו).

מהי הכתובת האבסולוטית?
הכתובת בהנחה שאני מתחיל ממקום 0.

מהי כתובת דינאמית?

הכתובת בה הנתונים שלי נמצאים, שהיא הכתובת אליה אני מנסה להגיע (בכתובת אבסולוטית) + הרגיסטר בסיס.

שקף 7-



כשאני מקמפל יש 90% בקשות system call, ואין לי מושג מה יהיה איתן. (??)

מה כל דבר עושה?

1. קומפיילר: מייצר תוכנית אסמבלי שעדיין לא ממופית בזיכרון – object.
2. לינקר (או פעולת הלינק): יש לי כל מיני ספריות שאני משתמש בהן, אז הוא מצרף אותן לתוכנית שלי.
   a) להבדיל מטעינה דינאמית - לא כל הספריות שאני משתמש בהן נכנסות לתוכנית שלי על ידי הלינקר, כי יש המון ספריות שהרבה מאוד פרוססים משתמשים בהן, והן נטענות רק פעם אחת לזיכרון, וכשאני כפורסס מגיע לזיכרון כדי לרוץ, מספרים לי איפה הן נמצאות.
   i) דפים משותפים - אותן ספריות שנמצאות בזכרון ומשותפות לכולן, נקראות shared pages.
3. ה- loader - בודק שהפרמטרים ששלחתי ל-system calls נכונים.

אחרי שלושת השלבים האלו יש לי אוביקט אחד שהכתובת בו היא אבסולוטית. אחרי שיש לי את זה ואני בא להריץ את התוכנית, בשורה הראשונה של האסמבלי אני מקבל את הערך של הרגיסטר בסיס (BR).

שקף 8 (אני = פרוסס)

אנו מבחינים בין שני סוגי כתובות:

1. כתובת לוגית – נניח שאני נמצא ב-CPU, ורוצה לגשת לכתובת . בתוך ה-CPU גם עובדים בכתובות אבסולוטיות (0), רק שיוצאים מה-CPU כדי לחפש משהו בזכרון, מערכת ההפעלה מוסיפה את ה-BR.
2. כתובת פיזית – איפה שהכתובת באמת יושבת. יש מערכות תרגום מסובכות בין לוגית לפיזית.

בזמן הקמפול, הכתובת הלוגית והפיזית הן אותו דבר.

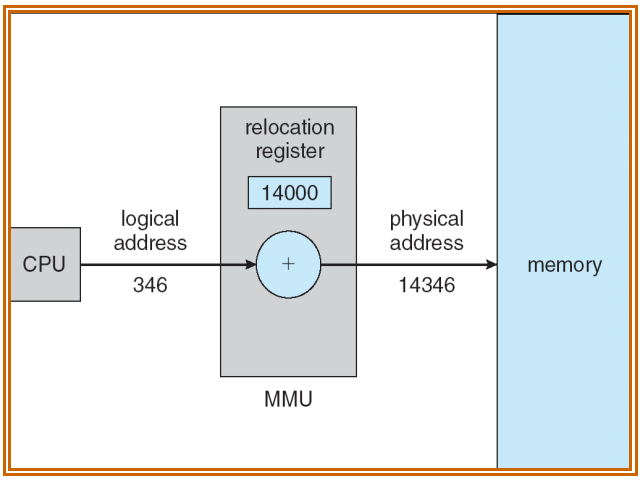
<u>שאלה בכיתה</u>:
מה היתרון בכתובת לוגית?
תשובה: אין פתרון, מתחילים מכתובת לוגית שהיא מתעלמת מכל הבלגאן בזכרון, כי כשאני כותב את התוכנית אני לא יודע מה יהיה בזכרון בזמן שאני ארוץ.

כשנותנים שטח ל-SWAP אנו לא רואים אותו יותר והוא עובר לשליטת ה- MMU. כלומר מבחינתו זה שקוף ואנחנו עדיין ניגשים כרגיל לכתובת האבסולוטית.

בעבר ה-SWAP היה בנוי מ-DRAMS.
בגלל שההארד דיסק היה מאוד איטי, לא היו יכולים להשתמש בו ל-SWAP. ה-DRAMS היו רכיבי זיכרון יותר מהירים (ויותר יקרים) מההארד דיסק.

<u>שקף 9</u>
יחידת MMU
ממפה לוגית לפיזית. (לפעמים תמפה לכתובת בזיכרון, הדבר קורה כי עכשיו שברנו לדפים. לפעמים יעשה מיפוי לדיסק הקשיח, אם הדף אליו מנסים לגשת נמצא כעת ב- SWAP)
כך, אם יש זיכרון בגודל 1 ג'יגה וגודל דף 4 KB, אני צריך טבלה בגודל של 250,000 ערכים.
זה מכניס סיבוך וזה יקר, ולכן יש חומרה שתאפשר חיפושים מהירים מאוד.
ה-MMU עושה את כל הטיפול בדפים – איפה נמצא כל אחד, הוא מחליף ביניהם וכו'.

<u>שקף 10</u>
אם ניקח מערכת של פעם, ויש לנו יוזר אחד, זו דוגמא למה MMU עושה:



מסתבר אבל שזה לא פשוט כמו שזה נראה..

<u>שקף 11</u>
כמו שאמרנו, נבדיל בין ספריה שאנו מכניסים לקוד שלנו בצורה סטטית (נעשה בשלב הלינקינג מיד לאחר הקמפול), לבין טעינה דינאמית, שזה נעשה רק בזמן הריצה. (רק בזמן שאנו קוראים לאותה פק' מהספריה המשותפת).

<u>שקף 13</u>
יש שתי דברים עיקריים שיש לקחת בחשבון:
1. אם דף לא נמצא בקש אבל כן בזיכרון

2. אם דף לא נמצא גם בזיכרון, ויש להביאו מהדיסק

בגלל השיטה החדשה שאנו מפרקים כל פרוסס לדפים, ניתן עכשיו שחלק מהפרוסס בכלל יהיה ב-swap, חלק בזכרון, חלק בקש.

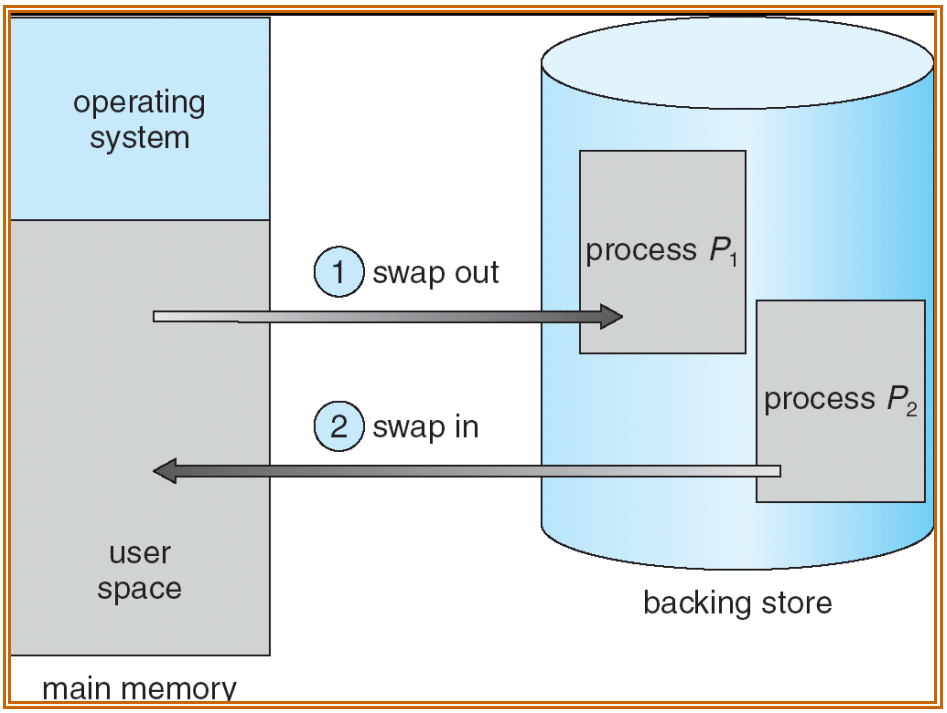
מנהל סיסטם צריך לקחת בחשבון את הדברים האלו ועלות ה-swap. זה חלק ממנגנון המתזמן.

<u>שקף 14</u>

אחרי שנעבור לשבירה לדפים, נוכל להעביר רק חלק מהדפים ל-SWAP, וחלקם יישארו בזיכרון. זה נקרא Paging in/out.

איך עובד ה-SWAP בעצם? (בשיטה הישנה, לפני השבירה לדפים):

כפי שנראה בציור (אזהרה! הציור עמוק):



<u>שקף 15</u>

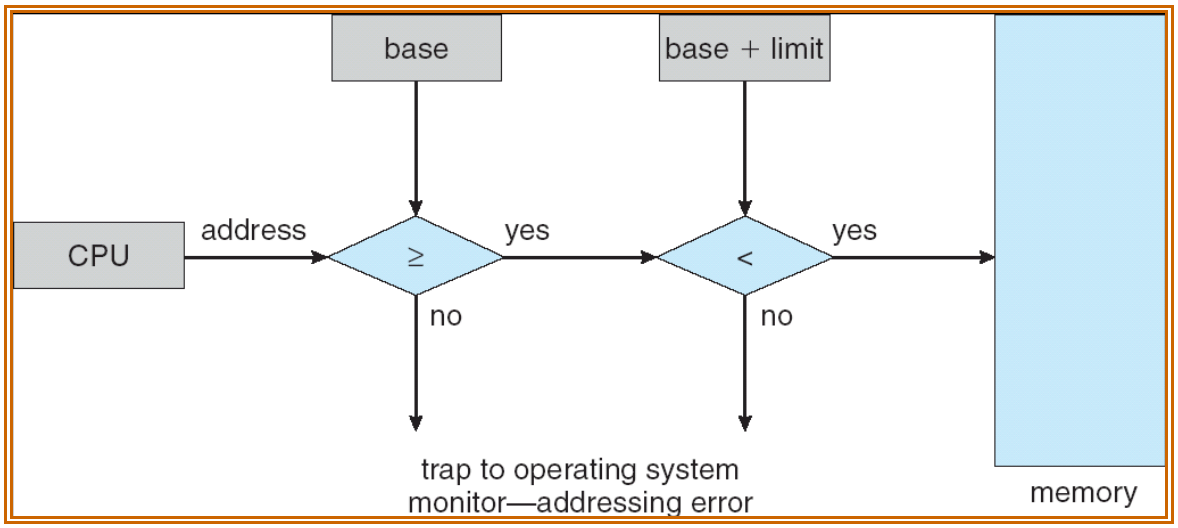
אין יותר הקצאות רציפות. (כלומר לא כל הפרוסס נמצא ברצף בזכרון)

<u>שקף 16</u>

נגיד שנרצה לכתוב מערכת הגנה טריוויאלית בין שני פרוססים:

אני בודק שפרוסס לא פולש לשטח של פרוסס אחר. ה-MMU דואג לכך.

ב-ready queue זה עדיין לא קורה, אלא רק בזמן ריצה.



שקף 17

אם מערכת ההפעלה רוצה לעשות הקצאות רציפות בכל זאת, יש כל מיני שיטות לעשות זאת (שוב, כאמור, לא עושים זאת).

מערכת ההפעלה מקצה בצורה רציפה, אם היא רוצה, והיא שמה פרוסס אחד בצורה רציפה בזכרון. אם בא אחד נוסף היא שמה אותו מיד אחריו, וכך הלאה כל הפורססים נמצאים ברצף אחד אחרי השני בצורה הרציפה. אם אחד הפרוססים יוצא מהזכרון (נגמרה התוכנית, הוא עף), ונוצר חור. כעת בא פרוסס אחר, צריך לחשוב איפה לשים אותו, ויכול להיות שהוא לא ייכנס בחור שיצר הפרוסס שיצא, או שהוא ייכנס אבל יותיר חור קטן יותר. חורים שנוצרים מורידים מאוד את יעילות החישוב.

כאן ניתן לראות את שלוש השיטות:
שיטות אלו מטפלות בחורים מכל מיני גדלים שנוצרים.

1. First Fit – החור הראשון שמתאים בגודל של הפרוסס (החור בגודל הפרוסס לפחות).
2. Best Fit – מחפש את החור הקטן ביותר אליו מתאים הפרוסס
3. Worst Fit – מחפש את החור הגדול ביותר אליו מתאים הפרוסס.

שקף 19

פרגמנטציה – איחוי.

שתי סוגי בעיות:

חיצוני: (נניח לרגע שאין MMU, שמקבלים גושי זיכרון רציפים) עכשיו לפרוסס יש כמות זיכרון מסויימת והיא רציפה. אם הוא ירצה עוד מקום, ויהיו חורים, יתכן שאני אבקש מקום שיש מספיק בשביל לתת לי (לחבר כמה חורים) אבל הוא לא רציף.
פנימי: לא מעניין, ולא במבחן.

שקף 20

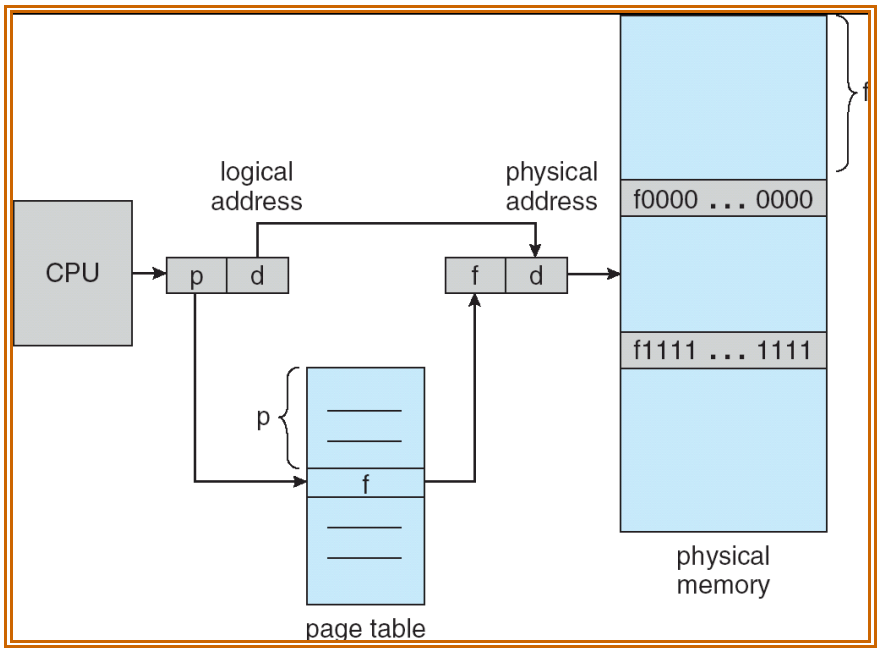
חלוקת התוכנית לדפים. המקומות בזכרון אליהם יכולים להיכנס דפים נקראים frames.
כלומר אנו מחלקים את הזיכרון ליחידות בגודל של פריים. הדפים נכנסים לפריימים.

שקף 21

כתובת שמיוצרת על ידי ה-CPU מורכבת משני חלקים:
כל כתובת בזכרון נמצאת בפריים – אז

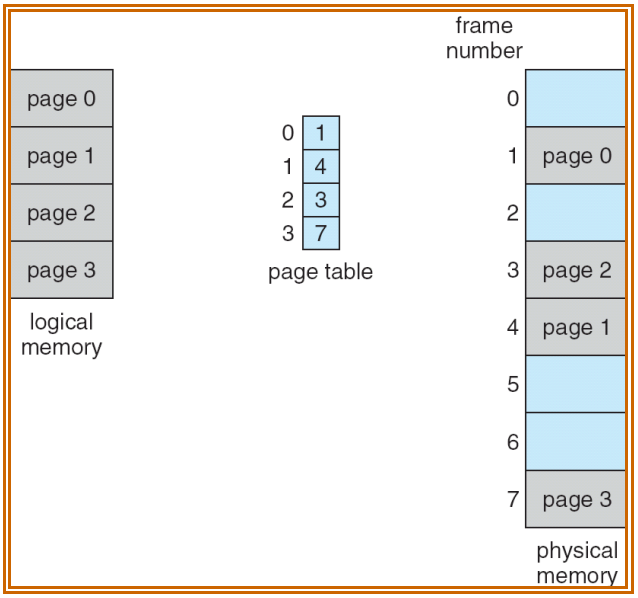
1. Page number – אומר לאיזה פייג' לגשת (יש לתרגם זה לפריים שבו הוא נמצא).
2. Offset – כמה לזוז בתוך הפריים הזה.

שקף 22
דוגמא:



ניתן לראות שה-CPU מחפש כתובת שמכילה page ו-offset.
דבר ראשון הוא הולך ל-pagetable, שם כתוב באיזה פריים נמצא כל page.הוא הולך לטבלה ומוצא בה את הפריים הרלוונטי, הולך אל הפריים הזה בזיכרון ומוסיף את האופסט.

שקף 23



יש לי תוכנית וה-MMU (לא אני!) שובר אותה לדפים.
זה מה שה-MMU עושה. הטבלאות שה-MMU מחזיק דינאמיות ומשתנות.
בתרשים בשקף זה ניתן לראות איך דפים מחולקים בין הפריימים ואיך יש טבלה שמאפשר לבדוק איזה דף נמצא באיזה עמוד.

שקף 24

כמו בשקף 23.

שקף 25

יש רשימה של דפים ריקים ואז מגיע פרוסס ואנחנו מקצים לו מקום. בניגוד למצוייר בתרשים, אנחנו יוצאים מנקודת הנחה שאין מצב כזה של דפים ריקים (הרי כל הנחת העבודה שלנו זה שהמערכת עמוסה מאוד). במצב כזה (שאין דפים פנויים), נצטרך לעשות replacement, וצריך לחשוב על אלגורתים שיעשה זאת בצורה הטובה ביותר. (שיעור הבא)

שקף 26

כאן ניתן לראות את אחד המימושים ל- page table: יש לנו 2 נתונים – איפה מתחיל (באיזה פריים) ומה הגודל. מכיוון שהטבלה ענקית והחיפוש בה יקר, אני רוצה לעשות לה סוג של קש, וכך אקבל חיפוש מהיר יותר. לכן עושים TLB, שהוא פשוט Look-up table מהיר (כי עושים בו חיפוש בו זמנית).
הסיבה שלא עושים את כל הטבלה כך מכיוון שקש זה דבר יקר).

שקף 27

מדגים את הרעיון של חיפוש page באמצעות המימוש מהשקף הקודם.

שקף 28

שתי סיבות בגללן יכול להיות miss לנסיון גישה לדף:
1. הדף לא קיים.
2. הדף לא בזיכרון (אלא בדיסק)

שקף 29

נדבר על זה יותר מאוחר – נחשב כמה עולה לי miss.

שקף 30

בכל פריים יש ביט הגנה, שאומר האם הדף בתוכו שייך לנו. דוגמא בשקף 31.

שקף 32

קוד משותף הוא בדר"כ משותף ולקריאה בלבד. דברים של ה- system שהרבה משתמשים בהם בם בדרך כלל כאלו. דוגמא בשקף 33.

שקפים 34-38

על מנת לייעל את החיפוש בטבלת הדפים, ניתן לחלק אותו להיררכיות ולחפש בכמה שלבים. דוגמא שראינו היא עבור שני שלבים.

ב- 37 ניתן לראות דוגמא: במקום לחפש ישר ב- 22 ביט, נחפש קודם ב- 12 ואז ב- 10. (page offset הוא מספר קבוע, במקרה זה 10 ביט, ולפיו ניתן לדעת את גודל הדף, במקרה זה 1K).
יש עוד שיטות יעול (שקף 38 ואילך) חיפוש עליהן לא דיברנו.

מצגת Virtual Memory

עד שקף 8 הכל די ברור מהמצגת.

<u>שקף 8</u>

איך נרצה להביא דפים החוצה ולהכניס אותם? נביא דפים כשנצטרך.
נשים לב שפה ה- invalid ביט שדיברנו עליו קודם חוסך, כי יתכן שימנע חיפוש על הדיסק (בלעדיו הייתי חושב שהוא לא בזיכרון, והולך לחפש אותו בדיסק) – דוגמא נפלאה לכך ניתן למצוא בשקף 11.

<u>שקף 9</u>

הרציפות בדוגמא זו היא רק בשביל לפשט.

<u>שקף 13 – דילג</u>

<u>שקף 14</u>

מה זה page fault? החטאה. לא מצאנו את הדף בזיכרון, וכעת יש ללכת להביא אותו מהדיסק. בשיעור הבא נראה כמה זה עולה.

שיעור 9

מצגת Computation of demand paging

אנחנו רוצים לעשות את החשבון כמה עולה לי page fault (אנשי סיסטם עושים חישובים יותר מדוייקים).

נחלק את הפייג פאולט ל-3 חלקים וניתן הערכה כמה כל אחד עולה:
1. הטיפול בזיהוי הבעיה
2. לקרוא את הדף מהדיסק
3. ה- CS שמתחיל את הפרוסס מחדש (הרי הוא יצא מה- CPU ברגע של page fault, מכיוון שכעת הוא מחכה ל- IO [הדיסק] ואין טעם שיבזבז את משאב ה- CPU).

אנו נראה כי הזמנים היקרים הם הזמנים המכנים (כלומר כאלה שדורשים את הדיסק לדוגמא), ופחות אלקטרוניים. זמנים מכניים הם בסדר גודל של ms (mili seconds) ואלקטרוניים של ns (nano seconds).

לכל דיסק יש controller, שהוא מערכת הפעלה קטנה שלו, שמחפשת על הדיסק מה שמבקשים.

שקף 6: ניתן לראות שנגרמה האטה של פי 250 מהמהירות הרגילה (כלומר מהירות בסדר גודל של ns).

שקף 7: אם נלך הפוך, ונתחיל בבקשה לכמה אנחנו מוכנים לירידה בביצועים, נגיד רק עד 10%, וכעת נחפש כמה פייג פאולט מותר לי (כלומר לקבל מה ההסתברות שצריכה להיות) נגלה שזה מספר מאוד קטן.

במצגת הבאה, שקפים 15-14, ניתן לראות עוד חישובים למיניהם, אך נשים לב שלא משנה איזה מספרים נקח – ברגע שעברנו ממהירויות של ns ל- ms המספרים עצומים ומקבלים ירידה גדולה בביצועים.

שאלה בכיתה: האם זה (פייג פאולט) תלוי במתכנת?
תשובה: גם. אם הוא מתרוצץ בזיכרון יש סיכוי טוב שהוא יגדיל את הפייג פאולט. בבית קשה לראות את זה, כי פה אנחנו מדברים על מערכת עם דרישות גדולות מאין ערוך מול משאביה (משרתת המון users עם דרישות שונות, וה CPU והזיכרון קטנים ביחד לזה). אנחנו נוכל לראות זאת בבית אם נכתוב תכנית שמשתמשת ביותר זיכרון ממה שיש לנו ונרגיש את ה- SWAP.

מצגת Virtual Memory

<u>שקף 25</u>

כעת חוזרים לדבר עם נושא שהתחלנו שיעור שעבר – כאשר רוצים להכניס פייג לזיכרון, אבל אין מקום וזה להחליט את מי להוציא.
אני צריך לקבוע מדיניות מי הוא הקורבן (ההנחה שלנו: כל הזמן צריך להעשות החלפות – אם יש הרבה פריימים חופשיים אז אין לנו בעיה, אלגוריתמי replacement באים כשכל הזיכרון מלא וצריך למצוא קורבן לפתרון הבעיה). בדרך כלל איך עושים זאת (מציאת הקורבן)? כותבים אלגוריתם ובודקים את הביצועים שלו על ידי סדרה אקראית.

<u>שקף 29</u>

אצלנו יש 3 פריימים בזיכרון. נראה את אחת השיטות: נשתמש כאן ב- FIFO (סוג של באפר מחזורים) ונראה מה קורה. רואים שיש 15 פייג פאולט – הרבה. אם אגדיל ל- 4 תאים, האם אוריד את מספר הפייג פולט? כן!

שקף 30

יש את האנומליה של בילדי (הממציא של virtual memory) שעדיין לא פתורה: הוא הראה דוגמאות שאתה מגדיל בהם את מספר הפריימים אבל לא מוריד את הפייג פאולט. בעיה זו נחשבת לאינהרנטית.
בשקף זה רואים את האנומליה ולא יודעים למה בדיוק זה קורה.
שאלה בכיתה: איך יכול להיות שזה עולה?!
תשובה: תקח דוגמא ותראה.

שקף 31

אלגוריתם ההחלפה האופטימלי = תחליף את זה שלא נשתמש בו בזמן הקרוב.
שוב, כמו כל אופטימלי, הוא לא פרקטי ודורש ידיעת העתיד.

שקף 32

באופטימלי רואים יש רק 9 פייג פאולט.

שקף 33

יש עוד 2 שיטות (מעבר ל- FIFO והאופטימלי):
1. LRU
2. FIFO – CLOCK עם ביט צאנס

CLOCK:

ה-clock - מוסיפים ביט אחד של second chance. יש גם אלג' עם פעמיים second chance.
דומה ל-FIFO (באפר ציקלי), אבל יש לו גם page used bit. אם ה-PAGE הוא רפרנס, ניתן לו את הביט הזה בתור אחד. אם קרה page fault, ויש לנו ביט שהוא 1, ניתן לו second chance. אחרי שנתנו לו, נעביר את הביט שלו ל-0. אם ה-use bit שלו הוא 0, נעשה לו replacement. כלומר בצ'אנס הראשון הוא הופך לאחד, בפעם השנייה הוא כבר אפס, ואז מחליפים אותו.

הגישה שלנו מאמינה שמשהו שאנחנו משתמשים בו הרבה, אפילו אם עשינו זאת מזמן, אז אנחנו נשתמש בו שוב (חישבו למה הדבר הזה מתאים לגישה שהצגנו)

דוגמאות לכל השיטות הללו, ניתן לראות במצגת More on Virtual Memory שקף 12.

CLOCK – מהדוגמא:

נסתכל על קלוק:
2 זה PF, 3-2 זה PF, עכשיו מגיע עוד פעם 2. במקרה הזה הוא ישנו. אין לנו במקרה הזה PF כי ישנו 2. עכשיו במקרה השלישי נכנס 1, יש מקום ריק אז אין בעיה.
בדוגמאות האלו השלושה הראשונים אינם נקראים fault אבל הם כן. כי פעם ראשונה שרצינו את 2 ולא מצאנו אותו, הכנסנו אותו לטבלה. גם מילוי הטבלה נגרם על ידי fault-ים. אמיר קורה ל-fault כאשר צריך למלא את הטבלה, אבל בדוגמא הזו ה-fault מוגדר רק לאחר שמתמלאת הטבלה.

עכשיו אנחנו ב-5 - איפה נכניסו?
אנו נוציא את 2 החוצה, כי היה לו כבר צ'אנס שני. 3 ו-1 עוד לא היה להם צ'אנס שני, יהיה להם רק ברגע שניגע בהם.
עכשיו הגיע 2, את מי אנו מחליפים? ברור שנחליף את 3 כי היה לו כבר צ'אנס שני.

נושא חדש – מערכות קבצים!

מצגת Unix File System

ברגע שיש הרבה תוכנות שצריכות את זה (גישה ושימוש בקבצים), רוצים שמערכת ההפעלה תטפל בזה.
אלה הדרישות שלנו ממערכת כזו.

File system – השיטה בה אנחנו מסדרים קבצים על הדיסק. נעמיק בהגדרה בהמשך.

נלמד מהו ה-file system הבסיסי:
יש לנו דיסק קשיח, עליו יש הרבה מקום.
אנו רוצים לדעת מה מקום פנוי, אנו אולי שומרים כמה דברים ורוצים לדעת מה אנו שומרים, רוצים אולי לנהל כמה FS- ים - CD, הארד דיסק, רשת וכו', יחד, רוצים לנהל אבטחת משתמשים, סימבוליק לינקס, לנהל שימוש בצורה אחידה על הדיסק ועוד.
את כל הדברים האלו נראה בצורה של FS.

<u>שקף 5</u>
הארד דיסק - איפה שמידע נשמר. לפעמים נשתמש בכמה הארד דיסק, לדוגמא בשביל אבטחת מידע או לביצועים. עבור ביצועים ניתן לעשות זאת לדוגמא:
אם ניקח את הדיסקים ונחבר יחד וכל פייל נכתוב חצי לדיסק הראשון וחצי לשני, אני אגיע למצב בו אני כותב יותר מהר כי אני מפעיל שניים בו"ז, במחיר של הגדלת הסיכון אובדן המידע - מספיק שאחד מ-HD ילך לאיבוד כדי שאני לא אוכל לשחזר את המידע.

<u>שקף 6</u>
יש FS אחד על כל partition.
במערכות אחזור מתקדמות יש לנו מערכי דיסקים בצורה של table ,raid, והם מאוחסנים במקררים. קורס הזה כאשר אנו מדברים על HD נדבר על פרטישן לוגי אחד עליו יש FS.
אנחנו לא נעשה הפרדה בין פרטישן ל-HD ל-RAID - מבחינתנו בקורס הכל אותו דבר.

<u>שקף 7</u>
אילו סוגים של HD יש לנו?

1. דיסקים מכניים - רובנו משתמשים בבית.
   a. יחסית גדולים, איטיים - לוקח להם יחסית הרבה זמן למצוא מקום ספציפי אותו הם צריכים לקרוא.
   b. לעומת זאת הם קוראים די מהר ברגע שהוא כבר במקום.
   c. נגיד נרצה ששה טראקים אחד אחרי השני, יהיה הרבה יותר קל ומהר למצוא אותה מאשר טראק 1, 1000, 7000.
2. דיסקים מסוג solid state (דוגמת דיסק און קי)–
   a. חדשני, יקר יותר
   b. יותר איטיים מזיכרון
   c. מעט יותר איטיים בקריאה רציפה מדיסקים מכניים
   d. הרבה יותר מהירים מ-HD ב-sig type.
   e. אין כל משמעות לסדר הקריאה.

f. למערכות סוליד סטייט יש בעיות אחרות של wear leveling - לדיסק און קי בשוני ממכני, יש מס' מקומות בהן אפשר לכתוב. אם תכתוב כל הזמן לאותו מקום, המקום יתבלה - דבר שלא יקרה בדיסקים מכניים.

3. עוד מקומות:
   a. CD ROM
   b. כונני טייפים.

<u>שקף 8</u>

FS - אחד משלושה דברים:
1. FS פיזי שיושבים לנו על דיסק (פרטישן שאו מחובר אלי למחשב או באופן חיצוני)
2. FS לוגי, כמו proc/ שמציג רשימת כל פרוססים שנמצאים במכונה.
3. Virtual file system כללי של לינוקס - אנו לוקחים כמה devices, ויכולים לפנות לכל אחד.

<u>שקף 9</u>

כמה הגדרות:
1. דיסק, פרטישן - איפה שאני שומר את הנתונים פיזית - לא מעניין אם מכנית, סוליד סטייט, CD וכו'
2. Mount - פעולה בה אני לוקח את ה-FS שנמצא על הדיסק ומצרף ל-FS כללי של לינוקס
3. Unmount - מה שקורה כשעושים EJECT לדיסק און קי. לא רוצה ש-FS יהיה יותר במערכת.
4. קובץ - בכל מקום שלא יצויין אחרת, הכוונה היא קובץ שממש כתוב על הדיסק.

<u>FS שנמצאים על הדיסק</u>

זה המבנה הבסיסי.
UFS - יוניקס פייל סיסטם.
מה שנאמר עכשיו אינו מדויק היסטורית, אבל המודל הבסיסי של UFS, הדרוש כדי להבין כל FS אחרים.
אחרים כמו מערכת הקבצים ext2, שכוללת את מבנה הבסיס שלו עם אופטימיזציות)
כמעט כל יוניקס שנפגוש, יש בו איזה FS שדומה ל- UFS עם אופטימיזציות.

אנחנו נתעלם משיקולים של איפה לשמור את הקובץ (??)

<u>שקף 12</u>

אז מה יש לנו ב- UFS? הוא בנוי על 3 דברים:
1. איפה שנרשום את הדטא (בלוקים בדיסק)
2. מצביע לקובץ, שמכיל אינפורמציה על הדטא (אחד לכל קובץ)
3. סופרבלוק: גוש נתונים שמכיל מה יש ב- FS הזה (בעיקרון אחד לכל FS)

<u>block</u>

איפה שהדטא נשמר. יש לו כתובת. קובץ יכול להימצא על כמה בלוקים ואין שיתוף של בלוק בין קבצים (אם בלוק שייך לקובץ, אפילו אם לא במלואו [במלוא הבלוק] – הוא שייך רק לו)

<u>Inode</u>

<u>הפניות ישירות</u> לקובץ חלק מהתוכן של ה-INDOE אומר שהקובץ נמצא בבלוק זה וזה, לדוגמא: הקובץ הזה נמצא על בלוקים 7,8,9. באיזשהו שלב לא ניתן להצביע על בלוקים ישירות? ה-INODE קטן, אז במקרה כזה מצביעים לבלוק שמכיל מצביעים לבלוקים אחרים, כלומר משתמשים ב: (קרא שורה הבאה)
<u>הפניה עקיפה</u> אומרת שבלוק 7 מכיל בתוכן שלו את מספרים הבלוקים שהפייל נמצא עליו.

Inode מכיל את כל המידע שמערכת ההפעלה שומרת על הקובץ – למעשה את מה שאנו מקבלים כשעושים STAT.
INODE לא מכיל את שם הקובץ! (שם הקובץ
לכל INODE יש מס' שהוא מזוהה יחודית.

מה בעצם ה-INODE מכיל?
כל מה שאנו מוצאים ב-STAT, טיימרים, מתי ניגשנו לקובץ, מתי שינינו אותו, מכיל הרשאות, מי היוזר, GROUP ID, הרשאות כתיבה, הרצה, רפרנסים ישירים ועקיפים, באיזשהו שלב גם הרפרנסים העקיפים הם הצבעה לבלוק של הצבעה לבלוק שמצביע לבלוק שיש בו את הקובץ. (על מנת לאפשר קבצים שיותר גדולים מגודל של (בלוק*(בלוק חלקי (גודל של מספר))).
אנו נמצא גם הצבעות שהן עקיפות בחזקה הרביעית.

<u>Super block</u>
קטלוג שאומר מה יש כרגע על הדיסק - מכיל מספר בלוקים, מס בלוקים פנויים ועוד מידע שתלוי באימפלימנטציה הספציפית של ה-FS.
מכיל גם את מספר ה-inode.

פי'צרים ש-UFS נותן:

נניח שדיסק אינו מוגבל, איך נדע מה הקובץ הכי גדול שניתן להעביר אחיו? מס' בלוקים שניתן להבציע אליהם כפול גודל הבלוק. נניח של-innode יש מס' שהוא אינט, 4 בתים, ובלוק האו 4k, אפשר להצביע על 1024 innode-ים בבלוק.

1. אם אני תומך בהצבעות עקיפות פעמיים - יש איינוד שמצביע על בלוק שמצביע על בלוקים, ניתן להגיע לקבצים עד 4 מגה כפול 1 K - כלומר עד 4 ג'יגה.
2. אם הצבעות עקיפות עקיפות - אפשר להגיע למספרים בלתי מוגבלים.
3. כל FS הוא מוגבל בגודל בכל מיני צורות - זו הדרך שלנו לדעת מה גודל המקסימלי. (לגבי FS בימינו, אומרים שניתן לשמור קובץ בכל גודל, אך הכוונה היא רק מבחינה פרקטית).

Filenames and directories
להסתכל בשקף.
כדאי לדעת איך עובדות הרשאות ב-directories (נקודה אחרונה בשקף):
ביוניקס, אם יש read permission ניתן לעשות LS, אם יש write permission ניתן ליצור קובץ חדש בדירקטורי.
אם יש execute permission, ניתן לעשות cd על התיקייה.

Hard links
זה הוא למעשה קובץ שמופיע בשתי מקומות שונים אך אותו הקובץ בדיוק. אלה הם שני inodes שמצביעים לאותם בלוקים. אם נשנה קובץ אחד גם השני ישתנה.
לדוגמא, gcc ו- g++ הם אותו קובץ! (הוא יודע מה לעשות על ידי כך שהוא בודק את [0]args)

מה קורה כשמוחקים HL? כלום. הקובץ נשאר – הבלוקים נשארים על הדיסק. הסיבה לכך היא כי בעצם על כל inode נרשום את מספר המצביעים הנוספים, כלומר מספר ההרד לינקים לקובץ. רק ברגע שנמחוק מצביע אחרון נמחוק את הקובץ. אם לא מחקתי את המצביע האחרון, אני פשוט מעדכן את כל שאר הפוינטרים שיש מצביע אחד פחות.

- Symbolic links
מה שאנחנו מכירים בשם shortcut (קיצור דרך).
קובץ שהתוכן שלו זה מצביע לקובץ אחר. מכיל כתובת שהיא הכתובת של מקום בה נמצא קובץ אחר.
תחת ההנחה שיש לי הרשאות קריאה, כשאנסה לקרוא את ה- SL, ה- FS יראה שזה מצביע לקובץ אחר וילך לקרוא אותו). יש רמה מסוימת של כמה רקורסיות כאלו ניתן לעשות, אבל זה יכול להיות יותר מפעם אחת - אפשר 4 פעמים לצלול לעומק.

Broken links
עם הארד לינק אנו באמת מצביעים על הקובץ ויודעים שהוא שם ויש גישה אליו.
ברוקן לינק הוא מצב בו סימבוליק לינק מצביע על קובץ אבל הוא לא שם – אולי שמנו את הקובץ לא טוב, אולי יש בעיה בהרשאות וכו'. אם נעשה LS נראה את הקובץ, ינסה לעקוב אחר הסימבוליק לינק, אבל הקובץ לא יימצא.
ברוקן לינק זה מצב בו ההצבעה שלי שבורה, מצביעה למקום שאין לי הרשאה לקרוא.
בהארד לינק אני תמיד מצביע לקובץ תקין, ב-soft link אף אחד לא יכול להבטיח לי את זה.
בבית - לנסות ליצור ברוקן לינק. ln עושה זאת. למחוק את הקובץ ולראות מה קורה.

Ideas form Berkeley
שם התחילו לחשוב על איך לשפר את UFS, בעיקר התעסקו עם מהירות.
גילו שאנו קוראים בלוקים לדיסק, ואז מוסיפים עוד בלוקים אבל בינתיים כתבנו פיילים אחרים והבלוקים מפוזרים על כל הדיסק. כשאנו קרואים את הקובץ אנו מתחילים להשתגע על הדיסק קדימה ואחרוה - זוהי פעולה מכנית יקרה ואיטית.
דבר ראשון הגדילו את הבלוק – ובכך הבטיחו ש-K4 יהיו רציפים (כי זה עכשיו גדול הבלוק).
(במקום 500 בתים 4000 בתים)

מה הבעיה בבלוק גדול?
אנו מנצלים תמיד בלוקים שלמים, אז קובץ אחד בייט יתפוס בלוק שלם. יש לי בלוק של 512 תפסתי שמינית מקום. בלוקים גדולים זה בזבזני. אז הוסיפו רעיון של שברי בלוק (fragment) - יתנו לחלק בלוק לפרגמנטים, לשמור אותם בצד. ברגע שקובץ גדל, נוסיף פרגמנט או שנעביר אותו לבלוק גדול יותר. (כלומר כן אפשר לשתף בלוקים ברמה מסוימת, אבל לא חופשי, יש הגבלות).
בעצם הדטה שיהיה עכשיו במקום אחר יהיה הקובץ ולא הפרגמנט - נחבר את הפרגמנטים ותמיד נעבוד באופן רציף.

מה זה פרגמנט?
בהנחה שאנחנו עובדים על הארד דיסק וקריאה רציפה זה הכי חשוב (לדאוג שדברים יהיו רציפים). אז אחת השיטות זה פשוט להגדיל את הבלוק ואז הגדלו את הרציפות של קובץ. אז הגדלנו את גודל הבלוק, ועכשיו ביזבזנו מקום (כי קבצים קטנים עדיין לוקחים בלוק), אז מה נעשה? לכל קובץ נרשה לו שהבלוק האחרון שלו יהיה בגדול חצי בלוק או רבע או שמינית בלוק, אז למעשה זה לא יהיה הבלוק האחרון כי הוא לא יהיה בלוק אז זה יהיה פרגמנט.

סופר בלוק -
גם הוא יכול להינזק. הם החלו לשכפל את הסופר בלוק בכמה מקומות על הדיסק כדי שיהיה יותר קל למצוא אותו. בנוסף הם הוסיפו פיצ'רים כמו סימבוליק לינק.

= Fragments
כאמור, איזושהי דרך לנצל בזבוז. אנחנו לוקחים את הבלוק, מחלקים אותו לחלקים, שומרים את החלק של ה-DATA בתוך הפרגמנט, ברגע שהקובץ יגדל נמחק את הפרגמנט ונעביר לבלוק פרטי שלם.

Catalog based file system
כל מה שלמדנו ביוניקס זה סוג של FS מבוססי קטלוג -יש סופר בלוק ממנו אני לומד על הקובץ. נמצא במקום מסוים שמערכת יודעת איפה (נגיד נמצא K1 אחרי הפיילים).
באופן עקרוני FS כאלו אנו נוטים לקרוא אותם מהר, מי שפעם ניסה לעשות mount לדיסק יודע שלינוקס עושה זאת בשנייה.
FS כאלו לעומת זאת נוטים להשתמש בנק' ספציפיות בדיסק יותר מאשר מקומות אחרים. כל פעם צריך לעדכן את הקטלוג - ועל כן באזור הזה של הדיסק אני משתמש (כותב מחדש) הרבה. ועל כן לא מתאים לסוגי דיסקים שצריך לפרוש את השימוש בו על מקום מסוים.
דיסק און קי - אין לו קטלוג, צריך לסרוק כל דיסק און קי בשביל לדעת מה קורה. הוא צריך לעשות את זה כי אין מקום ספציפי שאומר מה קורה. (מה שגורם לזמן ה- mount שלו להיות יחסית גדול)
נכון להיום, כל מערכות שנשתמש בהן, כוללות קטלוג. בווינדואס יש file location table, path32.
בלינוקס זה הסופר בלוק. במיינפריים זה הקטלוג.

הבעיות עם מודל של UFS (הבסיסי)
- אין לוג - אומר שפעולה היא לא אטומית, ואם היתה לנו נפילה באמצע לא ממש ברור מה יקרה, איך נשמור את הנתנוים. לוג זה אומר שאני רושם שאני הולך לבצע את הפעולה, אני רושם שביצעתי אותה, ואם היא מתרסקת באמצע אני יכול או לשחזר אותה או להשלים אותה. FS של UFS ידוע לרעה בזה שאם אין לו לוג הוא נוטה לאבד מידע ברגע שהיתה הפסקת חשמל לדוגמא, המחשב נפל באמצע כתיבה לדיסק.
- קבצים נוטים להיות שבורים \ מפוזרים \ fragmented - אנו מאבדים זמן על הזמן שבו הדיסק זז מצד לצד.
  - כאשר ברקלי עשו שינוי על גודל בלוק ופרגמנטינג הם הרוויחו 1000% בניצול קיבולת הדיסק. עד אז רוב הזמן הרוב הראש של הדיסק היה עובר מצד לצד ומבזבז הרבה זמן.
  - מערכות מודרניות מגיעות ל-80-70 אחוז של ניצול דיסק. רק מזה שמחברים פרגמנטים ביחד הגענו ליעילות שהיא פי 10 מקודם.
- אין ניצול מאוזן של הדיסק.

EXT2
ה-FS הישן של לינוקס, היום עובדים עם EXT3 שהוא הרחבה של זה.
EXT2 מכיל המון performance extension מעל ברקלי - מאפשר לנו לעבוד עם בלוקים בגדלים שונים. חוץ מזה הדיסק מחולק לכמה פרוסות של block groups. רק אם הקב' מלאה נבדוק מחוץ לקב'. משפרים לוקליזציה של קבצים.
כל פעם שרוצים כתוב לקובץ מקצים 8 בלוקים. אם מסיבה מסוימת לא נשתמש בהם - נפנה אותם. כך אם קובץ גדל עוד ועוד, תמיד הוא יגדל ברצף.
הוא גם מוסיף עוד תוספות כגון: שמות קבצים גדולים, FS שהוא מוגבל עד 4 טרה בייט, indirect^4, מוסיף את האפשרות להשאיר reserved space - נגיד יש לי סטודנטים באונ', אני מוכן לתת להם עד 95% מהדיסק, 5% אחרונים שמורים למנהלים.
הקוד של EXT2 מאוד קשה להבנה. כמעט לכל דבר יש כמה אופציות - מבחינה לוגית הוא עובד כגון UFS, אך הוא מוסיף הרבה דברים.

מבנה ה-innode של 2EXT

מה ההרשאות, מה היוזר ID, זמן יצירה וכו' ווכ'. (עוד הרבה בשקף).

הסופר בלוק שומר דברים שהם טריוויאליים ולא מעניינים. - בשקף.
יש innode bitmap, block bitmap, שהם מיפוי של תפוסים לפי ביטים
ביטמפ זה לעשות מיפוי של כל הבלוקים – כל בלוק מקבל ביט ואני מסמן האם הוא תפוס או לא.
למה? כי ככה מאוד קל למצוא אח"כ בלוקים פנויים.

אחד הדברים הבעייתי הוא שאין לוג -
באיזשהו שלב הוסיפו לוג ל-2EXT, זו התוספת המרכזית שהפכה את 2 ל-3.
כלומר על כל פעולת כתיבה אנו מציינים מה אנו הולכים לשנות, כותבים את זה ללוג, ואפשר להגיע לשחזור מלא של FS, גם אם הוא נפל באמצע.

- 3EXT
גם הוא וגם 4EXT מוסיף עוד יכולות נוספות לעומת 2EXT.
4EXT - FS ניסיוני.
הפיצ'ר המרכזי שהן מוסיפות לעומת 2EXT זה הלוג.

עוד FS שעובדים איתם בלינוקסי:
reiserFS - מאוד יעיל ומהיר. אומץ בחדווה על ידי קהילת הלינוקס, עד שהמפתח שהוא רצח את אשתו.
XFS
JFFS
CDRDFS - ה-FS של CDROM.

/proc

זו מערכת FS תמיוחדת שבעצם מכילה את כל הפרוססים (תיקייה לכל פרוסס) והמידע על הפרוססים שנמצאים במערכת ההפעלה (לדוגמא גודל מקסימלי של זכרון משותף). אם נעשה CD לכל פרוסס, נוכל לראות FDS הפתוחים אצלו ועוד מידע על קבצים. למה? כי ביוניקס הכל קובץ ומקל ביצוע של מוניטורינג.

doppler-10 12% cd /proc/
doppler-10 13% ls
1 13035 13226 151 2 2512 2974 3095 3233 3410 4080 6734 894 crypto ide loadavg partitions timer_list
10 13036 13238 152 200 25267 2982 31110 3245 3423 4084 68 895 devices interrupts locks sched_debug tty
1013 13062 13242 154 201 2618 2991 31124 3264 3447 4116 6925 9 diskstats iomem mdstat scsi uptime
1018 13089 13244 15508 202 26191 3 3119 3272 3450 4194 7 acpi dma ioports meminfo self version
1059 13169 13248 16902 203 2629 3010 3123 3323 3471 4207 71 asound dri irq misc slabinfo vmstat
11 13186 13302 1940 2290 26808 3036 3130 3339 3512 4208 72 buddyinfo driver kallsyms modules stat zoneinfo
11323 13194 13706 1941 23667 26836 3056 32144 3358 3815 5 8 bus execdomains kcore mounts swaps
12906 13195 13726 199 23687 27535 30867 32164 3360 3816 6 860 cmdline fb keys mtrr sys
13030 13199 13733 19980 2446 27671 30869 32165 3380 3924 67 862 config.gz filesystems key-users net sysrq-trigger
13031 13202 150 19985 2500 28744 30886 32166 3382 4 6730 887 cpuinfo fs kmsg pagetypeinfo sysvipc
doppler-10 14% cd 3245
doppler-10 15% ls
auxv cmdline cwd exe fdinfo loginuid mem mountstats oom_adj root smaps statm task
clear_refs coredump_filter environ fd limits maps mounts numa_maps oom_score sched stat status wchan

תרגול

מצגת first class in linux

אולי יסוכם בהמשך...

תרגול 9 (VFS)

מצגת unix file system, שקף 35 ואילך.

file system driver = FSD

הקדמה על VFS
כשאנו מגדירים סוקט וכו' אנו בעצם מגדירים קובץ שלא נטפל בו באמצעות המתודות של ה-FS שאנו נמצאים בו כרגע, אלא באמצעות מתודות אחרות.
הצירוף הזה של FS מסוגים שונים לתוך מרקם אחד זה ה-virtual fs של יוניקס.
אם נחבר כמה מדיות של FS (דיסק און קי, סי די רום וכו') - נקבל VFS.

המטרה - שכל דרייבר שאנו רוצים לתמוך בו יספק חתיכת קוד שיתמוך בכל המקומות שמשותפות לכל FS.
כשנקרא ל-mount וכו' - נקרא ל-VFS הממומש על ידי ה-FS Driver, והוא יעבוד שונה בהתאם לקובץ שאנו עובדים עליו כרגע.

לדוגמא אנו עובדים בכונן C.
נקרא ל-open, משהו שעובד מול ROM CD, אשר יש לה FS שונה לגמרי.
אם נקרא ל-disk on key, יש לו FS שונה, נקרא למתודה שמנהלת את ה-disk on key.

יש אינטר פייס גלובלי למה זה FS, לדוגמא - יש לו פונ' read_file אבל לכל דרייברים מממשים אותה בצורה אחרת.

פונ' שהיינו רוצים לראות:

- Vfs_mount – mount a file system
- Vfs_unmount – ubmount a file system
- Vfs_root – return the root vnode for the file system (what is vnode? Bare with me)

יחזיר איפה ה-FS בעצם מתחיל.

- Vfs_statfs – return file system specific info (answer to statfs(2))

נותן לי את כל האינפו' על FS - מתי עשו לה mount, כמה בלוקים יש בה, כמה בלוקים פנויים יש ב-FS וכו'.

- Vfs_sync – flush data to disk
- Vfs_fid, vfs_vget – beyond the scope (used by network file system)

מהו Vnode?
struct וירטואלי של הקרנל שהוא פוינטר לקובץ. (אם הקובץ ממומש על מערכת הקבצים UFS או משהו דומה, אז ה-vnode יצביע על inode. אם קובץ נמצא על ROM CD, יצביע על מבנה אחר.)

כשאנו מבצעים אופרציה, לדוגמא open, אנו עושים מניפולציה על vnode, וה-fS driver קורא את המניפולציה ומפעיל מתודה נכונה שמתאימה ל-FSD.
לא כל פעולות של vnode צריכות להיות ממומשות על כל FS.
לדוגמא - יש פעולות כמו סימבוליק לינקס שיכולות להיכשל על FS שונים.

<u>רשימה חלקית של פונ' שמוגדרות על vnode: זה הוקטור האופרציות של vnode</u>

- Vop_select – implement select(2)
- Vop_rdwr – read to or write from file
- Vop_link – implement link(2)

  פונ' שממממשת hard links.
- vop_rename – obvious
- Vop_mkdir – make directory
- Vop_rmdir – remove directory
- Vop_symlink – implement symlink(2)

<u>קצת דברים ספציפיים למערכות קבצים של לינוקס</u>:
לינוקס FSD ממומשים כ-kernel module.
FSD מודיע למערכת שהוא FSD, נרשם ברשימה, וכשאנו נעשה mount ל-FS, נציין באיזה סוג FS מדובר, וה-FSD הנכון יעלה.
ה-FS מממש את רשימת הפונ' שמטפלות ב-vnode וב-VFS שראינו קודם. מעביר למערכת הפעלה struct עם פוינטרים, זו הדרך לממש ירושה ב-C.
אנו מעבירים טבלה של פוינטרים לפונקציות.
ה-FSD מן הסתם יכיל את כל ה-API שאמור לטפל בדרייבר הספציפי שהוא תומך בו.
(בשיעור 10 (חלק שאינו גריד) יש הסבר בדיוק איך הדברים הללו נעשים)

עוד הערת אגב לא קשורה -
לינוקס תומך בנוסף לכל ה-FS שדיברנו עליהם, ל-FS שנמצאים ברשת. הוא תומך גם כן ב-FS מבוזרים - שנמצאים על כמה מחשבים, ניגשת אליהם קב' של מחשבים, מוצאים איזה קובץ הכי קרוב אלי מהקב' שאני רוצה ונותנים לי אותו בעצם.
חוץ מזה, אפשר לממש FSD בשני מקומות.

יש מערכת קבצים RAID, שמשתמשת בכמה HD-ים ומאפשרת לבצע גיבויים בקלות. לדוגמא: אם יש לנו שלושה HD-ים, אז בשניים נשתמש כרגילים, ובשלישי נשמור את ה-XOR שלהם. אם אחד מהשלושה יקרוס, נוכל תמיד לשחזר אותו באמצעות שני האחרים.

שיעור 10

נשאר לנו ללמוד: Grid (נעשה השיעור), קרנל ו- WINDOWS.

מה זה גריד?
מה שאני חושב – זה קלסטר
מה זה באמת גריד? לקחנו כמה ארגונים, מחשב אחד או יותר, לכל אחד מנהל אחר. המחשבים האלו, אנו משתמשים ב-idle time שלהם. מה שחשוב זה שלקחנו מחשבים בניהול שונה, בנינו ארגון גלובלי מכל הארוגנים, בהם יש משתמשים שונים עם ססמאות שונות וכו'. איכשהו בונים ארגון רויטואלי מכל הארגונים עם מיפוי בין משתמשים, סיסמאות משלו, בילינג בתור הארגון הוירוטאלי וכו' – מישהו שנמצא כאן ומתחבר לגריד יכול לשלוח רשימה, לצמצם עלות ביצוע, המתנה וכו' – יכולה לרוץ במחשבים באוניברסיטאות, התוצאות ייאספו, יהיה מיפוי בין המשתמש על שביצע משימה על הגריד למשתמש באונ', וכל הדבר הזה בסוף ייתן תוצאה. אם מערכת הפעלה היא על מחשב בודד שרץ בנפרד, גריד זה כמו מערכת הפעלה שרצה על הרבה מחשבים.
(אם נסתכל על הנושא המרכזי, אז קלאטר זו צורה עלשות עיבוד מקבילי שבו כל המעבדים אינם שקולים – בקלאסטר יש זיכרון לוקלי של כל תחנה, ..)
גריד מרחיב את הקלאסטר – אומר שיש מחשבים בהרבה סוגים והרבה בעלי בתים, לכל אחד דרישות אחרות ותקציבים אחרים וכו'. אם עד עכשיו OS התעסקה עם ציוד מסוגים שונים, גריד לוקח שלב אחד קדימה ואומרת שבגלל בעלי בתים שונים יש גם אינטרסים שונים. הקונדור זה כמו מערכת הפעלה.

<u>Grid</u>

מה הם השימושים של גריד (נכון, עוד לא הגדרנו מה זה)?

- בעיקר רפואיים, יש בעיות רפואיות שהגריד מתיימר לפתור:
  - סימולציות
  - קבלת החלטות ברפואה
  - בנית תמונה תלת מימדית [כשיש צורך רפואי בכך] (היום לוקח לעשות את זה עבור עצם בסביבות החצי שעה, דבר שלא ממש ישים בחדר ניתוח. לעומת זאת עם גריד ניתן לעשות זאת ב- 2 דקות)

<u>שקף 3</u>

מה רואים פה? פקק! יש לנו כאן את הבעיה הכללית של הרבה קליינטים (מכוניות) ולא מספיק סרברים (מערכת הכבישים). במ"ה פותרים זאת בדרך כלל על ידי תורים.

<u>שקף 5</u>

אם נסתכל נראה שה- HW לא מנוצל היטב, וזה נותן לנו כיוון למה נוכל לעשות. לדוגמא: (שקף הבא)

<u>שקף 6</u>

לקחת את העיבוד ולפצל אותו. SOA – ניקח משומיות טוריות ונעביר למחשב אחר **ייעודי לבעיה** (בגלל שהמחשב שלי הוא רב-תכליתי (multi-purpose) הוא לא יכול להיות מספיק יעיל). כדי שנוכל לעשות זאת צריך כמה מנגנונים:

- דפי זהב לשירותים
- איך אני מתקשר לשירותים (איך, מה התקשורת, מה הוא מבטיח וכו')
- תיאור של שירות הרשת (??)

שלושת אלו הם דברים אותם מגדיר ארגון שעוסק בתקינה הנקרא W3C.

<u>שקף 7</u>

גרטנר זו חברה שנותנת יעוץ לאיך לעשות את זה.
יש לנו:

- service provider (לא מספק את הכל בעצמו)
- לקוח

לדוגמא, בכל אתר (גדול הכוונה) יש שימוש באתרים אחרים – כמו באתרים שגובים ממך כסף, הם בדרך כלל משתמשים בשירות של אתר אחר (PayPal לדוגמא). כלומר אתרים שאנחנו מכירים הם למעשה תפירה של הרבה אתרים (מאחורי האתר יש עולם שלם של שירותים שמגיעים ממקומות שונים).

<u>שקף 8</u>
<u>עקרונות ארכיטקטורת גריד</u>
מבחינת הארכי' אנחנו מדברים על סביבה יציבה.
מה קורה כשמכבים מכונה של עיבוד מקבילי? אם היא קיבלה משימה לבצע - פחות אסון, גריד לא עובד אונליין. אם היה עליה DATA – לא טוב. יש פרוטוקולים של גריד שהמטרה שלהן ליצור אמינות על מערכות שאינן אמינות.
אם כל אדמיניסטראטור יצטרך לעבוד, ואנו מדברים על מערכת עם אדמי' רבים – זו בעיה. המערכת צריכה לעבוד ללא מגע יד אדם.

<u>שקף 9</u>
תומס אמר שבעתיד נצטרך רק חמישה מחשבים.
בהקשר זה יהיה נחמד לספר ש- Amazon מוכרים זמן חישוב. כלומר ניתן לשלם להם פר שעה של שימוש במחשב מסויים שהם נותנים לנו. הסיבה שהם עושים את זה כי חג מולד אחד היה עומס רציני באתר והם נאצלו לקנות הרבה שרתים חדשים. הבעיה היא שבשאר השנה אין להם ממש מה לעשות איתם אז הם מוכרים זמן חישוב שלהם לאנשים.

איך זה קשור לקורס?
עד עכשיו הייתה מכונה אחת שעשתה הכל. כעת ננסה לעבוד עם כמה מכונות (מה שיצור לנו בעיות חדשות)

<u>שקף 10</u>

כשאנחנו מגדילים (את כוח החישוב) יש 2 דרכים לעשות את זה:

- להגדיל את המחשב (יותר זיכרון, יותר CPU וכו')
  - היתרון של זה – לא מחייב שינוי
  - בעיות שנוצרות כאן:

- ההרחבה הזו מוגבלת (לא נוכל להוסיף עוד ועוד כמה שנרצה כי מתי שהוא לא נוכל עוד [כמו בדוגמא ב- PC שתומך בעד 10 ליבות])
- מחיר: גידול במחיר הוא מעריכי (נסו לקנות זיכרון, בהתחלה לא יקר יותר מדי, זיכרון קצת יותר גדול עדיין לא יקר, לא יקר, לא יקר, ... פתאום מאוד מאוד יקר!)
- עדיין אם המחשב נופל אני לא יכול לתת שירות.

לכן נוצרה התפיסה השניה:
- במקום להגדיל את המכונה נוסיף עוד מכונה!
- הבעיה:
  - צריך שהקוד ידע לרוץ על כמה מכונות
  - לא כל אלגוריתם ניתן למיקבול
- איך מתגברים על הבעיות?
  - לחלק את הפעולות לקבוצת עבודה (clusters), כל אחד מה יבצע פעולות שונות (לדוגמא אני אקנה אחד שעושה storage, אחד שעושה גרפיקה, אחד שמחשב מהר וכו'). <u>איך זה פותר את הבעיה?! זה לא לגמרי פתרון..</u>

<u>שקף 11</u>

נוכל לחלק כל תכנות לשני חלקים:
- אלה שנוכל למקבל (כמו סכום מספרים)
- אלה שחייבים להיות טוריים (כמו system calls)

חוק אמדל: מציג נוסחה חישובית מתקדמת שמראה עד כמה ניתן למקבל תהליך (אפשר לגרום לדברים לרוץ במקביל אבל לא עד אין קץ.

מיקבול מחייב שינוי האלג' של הקוד.

<u>שקף 12</u>

<u>יש כמה סיבות הלריץ קלאסטר:</u>
1. <u>יש מצב שאנו מפחדים שמחשב ייפול. זה נקרא high availability cluster.</u>
2. <u>לקבל ביצועים יותר טובים – load balancing</u>
   a) <u>לחלק מידע – יש ראוטר או FW שרוצים לבדוק את כל הפקטות שמגיעים לארגון – חצי ילכו לשרת אחד וחצי לשני</u>
3. <u>Compute cluster – יש חישוב ארוך מאוד לבצע, אחד יכפיל שורות זוגיות ואחד אי זוגיות</u>

Clusters – יש כמה סוגים:
- H.A:
  - תפקיד: להשיג זמינות כמה שיותר גבוהה על ידי:
    - Active-Active – שניים עובדים, ברגע שאחד נופל השני מטפל בהכל (ניתן לעשות זאת גם ל- Storage)

- Active-Passice – אחד נמצא במצב idle (לצורכי גיבוי) ברגע שהראשי נופל הוא נכנס לפעולה (לפעמים לוקח זמן לעשות את המעבר אבל הוא יחסית קצר)
    - Heart Beat - הדרך של שני מעבדים לדעת רביניהם מי חי ומי לו (הם שולחים אחד לשני"דפיקות לב")
- Load Balancing Cluster:
    - RR – בקשה ראשונה תלך למחשב הראשון, השניה לשני וכו'
    - ניתן לעשות זאת לפי פרמטרים אחרים כמו עומס ולאו דווקא RR.

יש לנו 2 סוגי קלסטרים:
- כאלה שעובדים בשיטה של תור (??)
- הדמיה של כל המחשבים למחשב אחד (כמו לחבר כל לוחות האם – בימיני אין דבר כזה שעובד טוב)

<u>שקף 13</u>

באוניברסיטה מותקנת **<u>מערכת קונדור</u>** שמאפשר לנו לעשות GRID, ומומלץ להשתמש בה (לשחק איתה קצת לדוגמא). איך משתמשים בה? בשקף מופיע סרקיפט בפורמט המתאים. סוג של מערכת הפעלה של גריד.
לדוגמא הערך של Queue אומר כמה עותקים של מה שכתוב כאן ליצור (לכל עותק נשלח פרמטרים אחרים וכל אחד ידע לחשב את משהו אחר [חלק אחר של התכנית לדוגמא] – כלומר אנחנו צריכים לכתוב את ה"מקביליות" [אנחנו אלה שכותבים את התוכנות והם פשוט מקבלות פרמטרים])
<u>שקף 14</u>

אז הנה גריד.
פעם היו חוות PC ואז אמרו אפשר לקחת כמה (חוות) ולחבר אותם יחד!
דוגמא לשימוש GRID גדול מאוד שנעשה בימינו הוא פרוייקט SETI (חפשו באינטרנט מה הוא בדיוק עושה!) שבשיא שלו היו 1.2 מיליון מחשבים שעשו עבורו חישובים!

שקף 15

חיבור קלסטרים בין מערכות:
(נתאר כל אחד המקום המתאים)

קח את הפרוסס ותשבור אותו ל- 2 תהליכים (Map & reduce או Map & accumulate, כמו שעשינו בסקים). Online לחלוטין! (?) נותן תשובה מהירה – מה שהצלחתי לחשב עד עכבר, כמו חיפוש בגוגל.	קח את המחשב, נצל את זמן ה- idle (כמו שומר מסך, חוסר בתזוזת עכבר וכו'). לדוגמא במחשבים במעבדות שלנו – הם מתחילים לעשות חישובים אחרי רבע שעה ללא תזוזת עכבר.	Dedicate- לוקח מחשבים ומייעד לעיבוד מקבילי	Batch-י לחלוטין. שולח עבודה, הולך לישון וחוזר אח"כ.

סה"כ – מה שחשוב זה שיש כל מיני צורות של איך מחלקים פרוסס לתהליכים:

1. צורה ראשונה – יש בפרוסס המון המון מיני תהליכים. המון המון פונקציות – רושמים על כל פונ' אם יכולה לרוץ במקביל או לא. אונליין. כאילו אין מנהל. כל מי שפנוי מריץ את המשימה, מה שלא פנוי נכנס לתורים. מצפה לקבל תשובה מיד.
2. אופליין – כמו קונדור, נותן משימה, מגיעה לקונדור, הוא מחפש מחשבים בזמנו הפנוי יישלח את זה וכו'. אחרי שנמצא את המחשב המתאים ביותר לפי דרישות החיוב, עיבוד וכו' נשלח לו את המשימה, היא תתבצע, בכפוף ללו"ז על אותו מחשב, מתישהו נקבל תשובה. אם יכבו מחשב אני אזהה את זה ואשלח למחשב אחר. אני אודיע לך כשהמשימה עובדת. מצפה לקבל תשובה בזמן מאוחר יותר.

שקף 16

Cloud – מחשוב ענן. להתחבר למקום ולקנות ממנו יכולות חישוב או Storage (אחסון) כמו שאמאזון עושים

שקף 17

<u>מה זה גריד?</u>

- מחבר בין מחשבים
- מחבר בין אנשים (ויש ריבים, מי מקבל מה)
- יש גרידי תקשורת
- שירותי חשמל

Foster & Kesselman נתנו הגדרה מדוייקת למה זה גריד (היא לא מקובלת על כולם):

- שיתוף בין משאבי חישוב שכל אחד עצמאי לחלוטין
- כל הסטנדרטים פתוחים

- צריכה לתת משהו לא טריוויאלי (כי אם היה אפשר לעשות את זה עם PC זה לא מעניין)

צריכה גם להיות on demand, והמשאבים צריכים להיות מרוחקים.

<u>שקף 20</u>

נחלק את המשאבים שיש לאירגונים וירטואליים. למשל יכול להיות שבנק דיסוקנט נמצא בארגון וירטואלי של בנקים ישראלים, וגם נמצא בארגון וירטואלי של איגוד בנקים בינלאומי וכו'. אפשר להיות בכמה גרידים, והם לא בהכרח חופפים.

<u>שקף 21</u>
מה בעצם הגריד מאפשר? לתת את צד שמאל של השרטוט באמצעות צד ימין

<u>שקף 22-24</u>
זה שרטוט שאנחנו מכירים. (כעת נוסיף לו עוד שלבים באמצע)
פתאום הגיעו רשתות, אז פתאום היה צריך לשנות חשיבה, וכאן נכנסים התורים – כמו בהדפסת רשת). אחרי זה הגיע ה- Grid Middleware שמאפשר לך להריץ אפליקציות מעל גרידים.

<u>שקף 25</u>

<u>חשוב להבין: מה זה accounting בגריד (בשביל בילינג), או איך עושים סבמיט</u>
<u>יש לנו את</u>:

- האיש – scheduler
- User – מזהה עבורו ושולח לו ג'וב, והוא ידאג לג'וב ויחזיר תשובה
- Catalog – אם השיתוף הוא של אחסון
- Info. Server – אני רוצה מחשב מסוג מסויים (או עם תוכנה מסויימת) והוא מוצא לי איפה יש כזה
- Computing Element – זה שמחשב
- Storage Element – איפה שהמידע
- Logging – מישהו שדואג לעשות לוג להכל. יש לוגר משותף לכולם בגריד.

<u>שקף 26-43</u>

עכשיו נראה איך המערכת (המתוארת בשקף הקודם) עובדת:
בתחילה נראה איך נראה ג'וב – תיאור של מה שאני רוצה לעשות. (28)
הוא מקבל זאת ומעביר לשרת הרשת שנותן אותו למנהל, ובאמצעות ה- Match Maker מוצא מקום שבו הוא יוכל להריץ אותו ("best" במרכאות כי זה כמובן תלוי בזמן ובתנאים).
לאחר מכן לוקח נתונים, מחזיר למנהל לאיפה כדאי לשלוח את הג'וב. הג'וב עובד שכתוב לשפה המתאימה (כמו condor וכו', כי כל מכונה יכולה לדבר בשפה אחרת) לאחר מכן מגיע ל- job control. הסיפור נגמר, ואז איש הלוג מתחיל לקבל נתונים.
JDL – job description language. השפה שבה כותבים סקריפטים.

שקפים 44-46

השקפים מתארים גישות שונות לאבטחה:

1. לא טוב
2. עוד יותר לא טוב
3. זה כבר סביר

שקף 47-50

ניהול הסקיוריטי – יש יוזרים של גריד, צריך לדעת שצריך לבצע מיפוי של היוזרים – איתי הגיש בקשה לקונדור, וחייבו את האוניברסיטה, איך גובים את הכסף ממנו. ברגע שמתבצעת פניה צריך למפות מישהו ליוזר בגריד. את המיפוי הזה צריך לשמור.
מה שצריך לדעת זה שיש יוזרים של אורגון, של ארגון וירוטאלי, הגריד צריך להיות מסוגל לעקבו אחרי היוזרים. מערכת הפעלה רצה מעל ארגון וירטואלי, זה גריד. אנו מאוד רגישים לדרישות אנשי סיסטם, כי לא עובדים באותן שעות ומדברים באותה שפה. מערכת הפעלה לקלאסטר פחות חשוב למעזר עבודה של סיסטם. בגריד חשוב שלא תהיה עבודה שתדרוש שיתוף פעולה, כי בגריד זה קשה. זה קשור להבטחה, כי איך נגדיר משתמשים לדוגמא?

המיפוי נעשה באמצעות certificate, באמצעות מפתחות public ו- private.
בשום זמן אנו לא שולחים את הסיסמא (אפילו לא מוצפנת)

הסרטיפיקייט הוא תוצאה של חישוב שמתחלף כל שעה, חישוב שאין לו פונ' הופכית, ולכן אי אפשר על סמך התוצאה (שניתן לפרסם) לדעת מה הקוד. אם לא אחליף את ה-, אני אנסה את כל הערכים ובסוף אמצא את הערך.

שקף 50

מתאר את מה שבאמת עושים כדי ליצור עוד סריאלים מהסריאל שלי (דבר זה נעשה כל 12 שעות ולכן אין לפורץ מספיק זמן, והמפתח כבר מתחלף) – וסריאלים נוספים אלו מאפשרים לוודא את סריאל האב אבל לא חושפים את הנתונים שלו.

שקף 51

באירופה, כמעט בכל המדינות של פרוייקטי גריד.
מה שהיה בשיעור הרוויח לראות תוכנה שמנטרת מערכת גריד באירופה וראינו תחנות חישוב ומעברי מידע בזמן אמת.

מצגת נצר: What does the Kernel do when

ביאור המושגים:

1. בלוק
   a) אפשר להסתכל על כל גוש זכרון כמערך ארוך. בלוק זה חלוקה 'דמיונית' של אותו מערך לקב' בגודל מסוים.
   b) למה זה יותר יעיל? כי כשאני קורא קובץ אז במקום שנצטרך לעשות המון קריאות לכל תא, ברגע שנחלק יהיו פחות קריאות.
   c) יש FS שעשויים מגדלי בלוקים שונים.
2. סופר בלוק
3. דיבייס
4. FS
   a)
5. MKFS
6. Mount
7. Minix
8. Umount
9. linking

מה זה Block Devices?
דיבייסים של O/I שה-DATA נשמר בצורה יעילה בצורה של בלוקים.

שקף 3

Mkfs.uxfs – userland program
כל MKFS הוא תוכנית שמקבלת דיבייס ומפרמטת אותו ל-FS מסוים (זה שבא אחרי הנק').
לדוגמא:

Mkfs.ext2

לדוגמא:

Mkfs.msdos

פתאום יש שינוי (אין רישיות בדוס לדוגמא).
מה זה אומר לאתחל?

1. ליצור סופר בלוק
2. להגדיר שאין שום קבוץ
3. להגדיר שהכל פנוי

יוזרלנד במובן שזו לא תוכנית לינוקס, אלא תוכנית ליוזר רגיל.

Register

אחרי שכתבנו FS צריך לרשום אותה (register), שלינוקס תדע שאפשר לעשות לו mount. כלומר זה נדרש רק פעם אחת (ל-ext2 מן הסתם לא צריך לעשות).
בטעינה כשאנו טוענים את המודול, יש את הפונקציה init שדילגנו עליו –
לכל מודול מגדירים פונ' init (או משהו בסגנון) ופונ' exit (או משהו בסנגון) שנקראות בטעינת המודול ובהוצאתו בהתאמה.

כשאני כותב FS אני פשוט עושה insmod, טוען כמודול. לינוקס לא יודע בפעם הראשונה איזה מודול זה – הוא מטפל בסאונד נגיד?
אבל הוא תמיד יקרא ל-init שלי בטעינה. (lsmod מדפיסה את המודולים הטעונים כבר).
כשנקרא לפונ' init בפעם היחידה, אז יהיה register.
כל פעם שאני עושה insmod – נרשמת.

<u>מה ההבדל בין insmod ל-init</u>?
Insmod זו הפקודה שאנו מקישים ב-shell שטוענת לי מודול, ו-init זו פונ' של המודל שנקראת בטעינה. (במסגרת ה-insmod נקרא ה-init).
יש שיטה איך נרשמים כ-FS (צריך להסתכל בקוד, לינוקס מספקת API, יש API שמשתמשים בו כדי להירשם כ-FS).

כשעושים insmod למודול שהוא מסוג FS, אז כמו לכל מודול קוראים לפונ' init שלו. אבל בגלל שזה מודול מסוג FS, הפונ' INIT שלו עושה סוג של הרשמה בתוך מנהל ה-FS של לינוקס.
הקרנל (המנהל) שומר רשימה של כל מנהלי ה-FS (FS handlers), מודולים שמטפלים ב-FS.

הקדמה לשקף: איך עושים object oriented ב-C:
ב-C כידוע אין עצמים, יש רק struct-ים שהם מבנים שמכילים מידע אך לא פונקציות כמו אובייקטים.
בגלל ש-C משתנה יכול להחזיק בתוכו גם פונקציה, אפשר לנצל את ה-struct-ים כדי שבאיזשהו מקום יחזיקו פונ'.
בסי, שם הפונקציה הוא מצביע לפונקציה (בדומה למערכים)
דוגמא:

```
Int func(char a){
        Print ("hello" + a);
}
Struct Cat{
        Int age;
        Int (*function)(char);
}
Cat a;
a.age = 5;
a.function = func;
a.function("!");                → prints: hello!
```

הקדמה: בשביל להבין טוב יותר את השקפים הבאים, נסתכל על ה-struct של סופר בלוק של לינוקס:

```
struct super_block {
        struct list_head s_list;          /* Keep this first */
        kdev_t                  s_dev;
        unsigned long           s_blocksize;
```

```
unsigned char             s_blocksize_bits;
unsigned char             s_dirt;
unsigned long long        s_maxbytes;      /* Max file size */

struct file_system_type   *s_type;
struct super_operations   *s_op;
struct dquot_operations   *dq_op;
unsigned long             s_flags;
unsigned long             s_magic;
struct dentry             *s_root;
struct rw_semaphore       s_umount;

struct semaphore s_lock;
int                       s_count;
atomic_t         s_active;

struct list_head s_dirty; /* dirty inodes */
struct list_head s_locked_inodes;/* inodes being synced */
struct list_head s_files;


struct block_device       *s_bdev;
struct quota_mount_options s_dquot;       /* Diskquota specific options */

union {
        struct minix_sb_info     minix_sb;
        struct ext2_sb_info      ext2_sb;
        struct hpfs_sb_info      hpfs_sb;

        struct ntfs_sb_info      ntfs_sb;
        struct msdos_sb_info     msdos_sb;
        struct isofs_sb_info     isofs_sb;
        struct nfs_sb_info       nfs_sb;
        struct sysv_sb_info      sysv_sb;
        struct affs_sb_info      affs_sb;
        struct ufs_sb_info       ufs_sb;

        struct efs_sb_info       efs_sb;
        struct shmem_sb_info     shmem_sb;
        struct romfs_sb_info     romfs_sb;
        struct smb_sb_info       smbfs_sb;
        struct hfs_sb_info       hfs_sb;
        struct adfs_sb_info      adfs_sb;
        struct qnx4_sb_info      qnx4_sb;

        struct reiserfs_sb_info  reiserfs_sb;
        struct bfs_sb_info       bfs_sb;
        struct udf_sb_info       udf_sb;
        struct ncp_sb_info       ncpfs_sb;
        struct usbdev_sb_info   usbdevfs_sb;
        struct cramfs_sb_info    cramfs_sb;
        void                     *generic_sbp;

} u;
/*
 * The next field is for VFS *only*. No filesystems have any business
 * even looking at it. You had been warned.
 */
struct semaphore s_vfs_rename_sem;        /* Kludge */

/* The next field is used by knfsd when converting a (inode number based)
 * file handle into a dentry. As it builds a path in the dcache tree from
 * the bottom up, there may for a time be a subpath of dentrys which is not
 * connected to the main tree.  This semaphore ensure that there is only
   Ever
 * one such free path per filesystem.  Note that unconnected files (or
   other
 * non-directories) are allowed, but not unconnected diretories.
```

```
    */
    struct semaphore s_nfsd_free_path_sem;
};
```

<u>שקף 5 – ext2 example and source</u>

```
struct file_system_type
{
struct super_block *(*read_super)(struct super_block *, void *, int);
```
// משתנה שאנו שמים בו את הפונ' שמקבלת מצביע לסופר בלוק ומספר, ומחזירה מצביע לסופר בלוק.
```
char *name;
int requires_dev;
struct file_system_type *next;
};
```
// עד כאן הסטראקט הבסיסי של מערכת קבצים בלינוקס

```
#ifdef CONFIG_EXT2_FS
register_filesystem(&(struct file_system_type) {ext2_read_super, "Ext2", 1, NULL});
#endif
```
// בשורות אלו אנו רושמים את מערכת הקבצים EXT2. אנו קרואים לו עם File_systme_type בו מילאנו את הנתונים עבור ext2.

<u>הסבר</u>: (שקף 6)

בכל mount הפונקציה קרואת את הסופר בלוק, מקבלת את המבנה הכללי של הסופר בלוק של לינוקס על מנת למלא את האופציות של ה-mount של ה-FS הספציפי שלנו (כ-void pointer).
המספר int הוא רמת הדיבוג (debug verbosity index).

- Char* name = שם של מערכת קבצים שאנו יוצרים. כאשר נותנים t name– הוא יחפש את מערכת הקבצים שלנו.

לדוגמא:
חיברתי disk on key, ונגיד שהוא נמצא ב-
/dev/dok
עכשיו נרצה לעשות לו mount, ואנו יודעים שמערכת הקבצים שהוא עובד איתה הוא fat לדוגמא, נעשה לו mount בצורה הבאה:

```
mount /dev/dok –t fat
```

אז ה-char* name יהיה במקרה הזה fat.

- Requires_dev – משתנה בוליאני, true / false, יקבל true אם אנחנו עובדים על מערכת קבצים מבוססת דיסק (בשונה ממערכת קבצים מבוססת אינטרנט לדוגמא).
- Next = המצביע למערכת קבצים הבאה – לינוקס שומר את fs handlers ברשימה מקושרת.

<u>שקף 7</u>

מה קורה כשעושים Mount ל-FS?

כשעושים mount הפונ' read_super (משקף 5) נקראת. זה לא באמת פונ', אלא משתנה שמחזיק בתוכו את הפונ'. לדוגמא, בשקף 5, שמרנו בתוכו את הפונ' ext2_read_super.
הפונקציה:
struct super_block *(*read_super)(struct super_block *, void *, int);
מחזירה מצביע לסופר בלוק,שזה גם מבנה של לינוקס.
כשעושים mount קוריאם לפונ' הזו, והיא ממלאת את ה-struct של הסופר בלוק.
אותה פונ' בודקת וממלאת את ה-**magic number** (לסופר בלוק יש מספר שהוא אמור להיות שונה בין מערכת קצבים אחת לשנייה. כשאנו עושים mount ל-device אנו בדוקים אם ה-magic number של הדיבייס זהה לזה של ה-FS. כשעושים mkfs על הדיבייס, כחלק מהגדרת הסופר בלוק, מוגדר בו ה-magic number.)

<u>את זה עושה read_super:</u>

- מגדירה גודל הבלוק(s_blocksize)
- ממלא את וקטור ה-operations (הסבר בשקף 9) עבור ה –FS המדובר. בתוך הסופר בלוק יש struct של operations שיש בו הרבה פונקציות.
- ממלא את הרפרנס לספרית ה-ROOT (/) של ה-FS (s_root).

כשעובדים על בלוק דיבייס גם נקראת הפונ' set_block_size.

- מרבית ה-UFS גם יקצו מבנה נתונים בשביל ה-root של ה-FS.

<u>שקף 8</u>

קריאת הסופר בלוק

על ידי קריאה לפונ' bread, אנו קוראים <u>בלוק</u> מהדיסק.

<u>איך bread עובדת</u>?

אנו שולחים לה את מה הדיבייס מתוך הסופר בלוק כפי ששומר זאת הסופר בלוק, בלוק ID שאנו רוצים לקרוא, ואת גודל הבלוק.

- Bh=Bread (sb-s_dev, block_id, sb->block_size)

הקריאה לפונ' תחזיר buffer handler שמלא ב-DATA, שניתן להגיע אליה דרך השדה b_data שנמצא בתוך ה-buffer handler.

- צריך לשחרר בלוקים אחרי שקראנו אותם באמצעות הפונ' breleased. אחרת, אם נקרא פעם נוספת ל-bread על אותו בלוק, מבלי ששחררנו אותו, לא נצליח.
  - הקרנל עושה המון נעילות.

שקף 9 – super block operations
כך נראה וקטור הפעולות שסופר בלוק מכיל (נשמר ב-s_op)

```
struct super_operations
{
void (*read_inode) (struct inode *);
int (*notify_change) (struct inode *, struct iattr *);
void (*write_inode) (struct inode *);
void (*put_inode) (struct inode *);
void (*put_super) (struct super_block *);
```

```
void (*write_super) (struct super_block *);
void (*statfs) (struct super_block *, struct statfs *);
int (*remount_fs) (struct super_block *, int *, char *);
};
```

בשקפים הבאים נעבור על רובן.

<u>שקף 10:</u>

הפונ' Write_super

בה משתמשים כדי לשמור את הסופר בלוק על הדיסק עצמו. לא בטוח שהשמירה תהיה קונסיסטנטית למה שקורה ב-FS (לאור דברים אחרים של לינוקס כגון cache).
אנחנו לא עוצרים את הקרנל בשביל ה-IO, וכתיבה היא פעולה כבדה. כשכותבים לדיסק זה לא באמת כותב לדיסק, אלא לבאפר. זה נכתב רק כשעושים flush, וזה על ידי put_super(sb).
אם נעשה read אחרי write, נקבל את המידע המעודכן, יכול להיות שהוא יקרא את זה מבאפר ולא מדיסק. ואז אם נוציא את הדיבייס מבלי לעשות unmount, אז המידע יהיה בעייתי.

<u>שקף 11</u>:

הפונ' put_super(sb)

ה-VFS קורא לפונ' הזו כשעושים unmount ל-FS, היא אמורה לשחרר את הסופר בלוק ואת המידע. מבטיח קונסיסטנטיות ומוחק.

<u>שקף 12</u>

Statfs(sb,statfsbuf)

כמו כל שאר הפונ' בוקטור מקבלת סופר בלוק, וגם באפר של statfs.
מה עושה? הפונ' הזו תמלא את ה-statfsbuf (שהוא מסוג struct statfs) ששלחנו לה במידע על הדיסק (לדוגמא מס' בלוקים חופשיים, גודל בלוק ודברים כאלו).

<u>שקף 13</u>

Remount_fs(sb,flags,options)

עושה mount מחדש עם דגלים ו-options אחרים.

<u>שקף 14</u>

Read_inode (inode)

הפונקציה קוראת unode פיזית מדיסק, היא שקרואת את fs_inode. מביא ת fs_inode, הכללי , מוציא unode number מתוך הכללי, בדוק שחוקי. מוצא מס' בלוק לתוכו צריך לקרוא, עושה s_b_btrsf, ומחזיר buffer head. יש ux_inode שהוא bh data.
Sb_bread היא פונ' שהולכת לסופר בלוק, מוציא בלוק דיבייס, וכן את הגודל של הבלוק.

בקרנל 2.6 הופך להיות xxiget.
יש inode כללי, ויש inodeפרטי. המידע על מה קיים על הדיסק נמצא בפרטי.
במה?

העיקרון – יש stuct inode של מערכת הפעלה, ויש ext2 inode. הוא אמור לרשת אבל אין ירושה.
אפשרויות:

1. אופציה ראשונה, בה הקרנל תומך וזה מה שהתכוונו – ששניהם יצביעו אחד על השני. זה מה שנראה בקוד של הספר, כמצביע ל-i_private.
2. אופציה שנייה – fs inode מכיל את struct inode. זה מה שעשו ב-ext2. יש דברים שניתן לחשב את האופסט שלהם, ה-struct inode תמיד יתחיל בדלתא מסוימת מ-fs_inode. נותנים ללינוקס את הכתובת של struct inode. את זה רואים בשקף hoe ext2 allocate inode.

ההבדל בין השתיים –

(1) שני פוינטרים יפה וברור – אבל אם אתה עושה malloc פעמיים זה יכול להיות על דפים אחרים, ואולי תצטרך פעמיים לקרוא לקש.

(2) מה שטוב באופציה השנייה – אופטימיזציה ביצועים, תהיה אלוקציה על עמוד אחד, חוסך fetch מהזכרון.

<u>שקף 15</u>

Notify change(inode, attr)

במרבית ה-FS היא אינה ממומשת.

מה עושה? נותנים לה inode, והיא תודיע לי כשמשהו קורה לו.

<u>שקף 16</u>

Write_inode(inode)

יש מידע פרטי, נגיד הבלוקים indode שהועבר לה על הדיסק. ל-inodeשומרת את המידע של ה-שמחזיקים את הקובץ.

<u>שקף 17</u>

Put_inode(inode)

הפונ' נקראת על ידי הפונ' ()i_put אם אין יותר צורך באיינוד.

המטרה שלה היא למחוק את הקובץ באופן סופי ולשחרר את הבלוקים שלו, **אם אין לינקים לקובץ**.

<u>שקף 18</u>

Reading the root I-node – ממנו אני אגיע לשאר הקבצים.

דורש קריאה ל-d_alloc_root (מתודה של הקרנל), ותלויה במימוש של שמירת ה-file_operations, של פעולות הבלוק, פעולות ה-inode.

<u>שקף 19</u>

כתיבת הסופר בלוק לדיסק

כאשר עושים unmount ל-FS, אז צריך לכתוב את הסופר בלוק לדיסק. יש פונ' מתאימה שנקראת לצורך כך. היא לא עושה שום דבר מיוחד, חוץ מלרוקן את הבאפר לתוך הדיסק.

<u>שקף 20</u>

קריאת תיקיות

באמצעות readdir(3) או הפונקציות שלו.

בסופו של דבר מי שנקרא זה getdents(2).
התיקייה מכילה רשימה של file_name-ים ואת ה-i-node-ים שלהם.
אנקדוטה קטנה – כשעושים ls אז הפונ' readdir נקראת מספר פעמים.

שקף 21 –
Struct Dirent

int getdents(unsigned int fd, struct dirent *dirp, unsigned int count);
dirent הוא struct שמחזיק בתוכו קובץ תיקייתי אחד (כלומר זוג של שם קובץ ואיינוד), ומצביע לבא.

```
struct dirent
  {
    long d_ino;              /* inode number */
    off_t d_off;             /* offset to next dirent */
    unsigned short d_reclen;   /* length of this dirent */
    char d_name [NAME_MAX+1];  /* file name (null-terminated) */ }
                  // מחזיק אורך של שם הקובץ, ו-1 + name max, זה בשביל ה-\0 בסוף
```

שקף 22

File name lookup

איך אנו בעצם בודקים שקובץ קיים?
אנו סורקים את כל הדברים בתיקייה, ואנחנו בסוף נמצא את הקובץ אם הוא נמצא או לא. אם אנו לא נמצא את הקובץ אנו מחזירים EACCES, אם נמצא נחזיר inode.

איך היא מיושמת?
יש סטראקט שנקרא inode_operation (כמו קודם, struct שמכיל הרבה פונ' שעושות דברים על inode).
פונ' ה-lookup נמצאת בתוך הסטראקט הזה, והיא זו שמחפשת את הקובץ.

שקף 23
FILE I/O
פעולות file I/O מתבצועת באמצעות הוקטור file_opration. יש 4 פונ' באמצעותן אנו עושים פעולות IO:

- **Lseek**
- **Read**
- **Write**
- **Mmap** (ממפה קובץ מהדיסק לזכרון. היא גם קוראת וגם כותבת – קוראת מהדיסק ושמה את זה בזכרון ואז אפשר לשנות את הזכרון. כאשר כותבים munmap היא מעדכנת את הדיסק לפי השינויים שעשינו בזכרון).

פרט למקרים חריגים, בדר"כ lseek,read, write יעבדו אותו דבר, לא משנה באיזה FS נעבוד.
ברוב- ה-FS ה-file system operation יהפוך להיות ברירת המחדל של לינוקס, יש מצבים בהם הוא יאותחל להיות משהו אחר.
בתרגיל ניתן להשתמש בדיפולט של לינוקס.

שקף 24
Memory mapped files
נעשה באמצעות אינטרפייס כללי.
יש וקטור של פונ' שמטפל בזה, בשם address_space_operations.
בוקטור הזה יש פונ' של קריאה מדפים, כתיבה של דפים, סינכרון, שינויים וכו'.
כמו ב-file_ops, גם כאן לינוקס מספקת דיפולטים.

חשוב מאוד להבין שהמבנה של FS בלינוקס וביוניקס, הוא מבנה שבעצם לא מניח שום דבר או כמעט שום דבר על ה-FS או על המדיום בו ה-FS כתוב. יש FS שרוצה לעבוד מהדיסק, אז נממש read inode וכו' שעובד מעל דיסק. אותו דבר מעל רשת וכו'.

כל דבר ב-FS של יוניקס אפשר להחליף, הכל עובד באמצעות פוינטרים לפונ' שתלויות במקרה הספציפי שאנו מממשים.

<u>שקף 25</u>
קריאה ל-stat
Statfs(2) היא פונ' שמחזירה מידע לגבי ה-FS. כשקוראים לה, לינוקס הולכת וקוראת לפונ' statfs של ה-FS המתאים, FS_name_sstatfs(). אותה פונ' מגיעה מהוקטור של ה-super_block_operation. (ניתן לראות בשקף 9).

שקף 27:
טיפ ל-C: איך ליצור struct ועל הדרך להציב בו נתונים:

```
struct file_operations ux_file_operations = {
        llseek:    generic_file_llseek,
        read:      generic_file_read,
        write:     generic_file_write,
        mmap:      generic_file_mmap,
};
```

שקול ל:

```
Struct file_operations ux_file_operations;
ux_file_operations.llseek = generic_file_llseek;
ux_file_operations.read = ….
```

עוד דוגמא:

```
struct address_space_operations ux_aops = {
        readpage:         ux_readpage,
        writepage:        ux_writepage,
        sync_page:        block_sync_page,
        prepare_write:    ux_prepare_write,
        commit_write:     generic_commit_write,
        bmap:             ux_bmap,
}
```

<u>שקף 28</u>
השמות

```
inode->i_mapping->a_ops = &ux_aops;
```

הסבר: לאיינוד הספציפי הזה, אני הולך ל-i_mapping שלו, והוא מכיל את וקטור הפונקציות, ואני שומר בתוכו מצביע לוקטור הפונקציות הספציפי. במקרה שלנו הוא נקרא ux_aops.
כל פעם שניצור קובץ חדש נגדיר זאת עבורו.
עוד דוגמאות:

```
inode->i_fop = &ux_dir_operations;
inode->i_fop = &ux_file_operations;
```

לקרנל החדש סארקטים שונים.

<u>שקף 29</u>
הקרנל החדש, 2.6 – מבנה האיינוד שונה. אותה לוגיקה – אבל שמות מבנים אחרים. לדוגמא:

```
struct inode {
        struct inode_operations *i_op;
        struct vm_area_struct   *i_mmap;
        ....
        };
```

<u>עד כאן סיכום החומר.</u>

השיעורים של וינדווס והשיעור האחרון לא מסוכמים ולא במבחן.

ישנם עוד 3 תרגולים (האחרונים) שלא מסוכמים, הם מכילים הרבה קוד וצריך לעבור עליהן.