

תרגול 1

מטה-דין על הקורס "מערכות הפעלה"
מהי מערכת הפעלה?
עקרון הוירטואלייזציה
מערכת הפעלה "לינוקס"

מזה-דיון על הקורס "מערכות הפעלה"

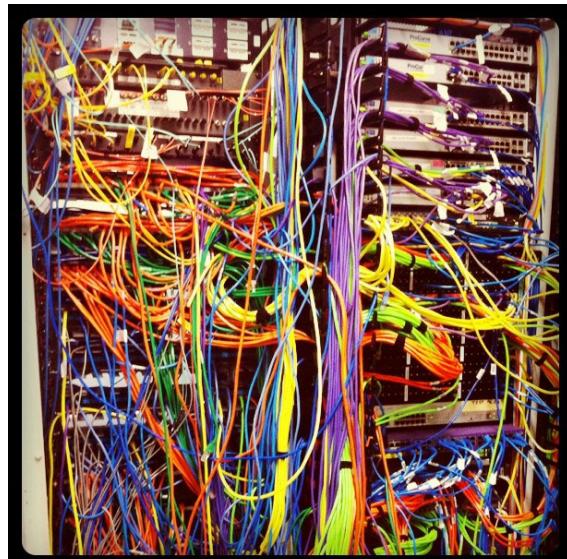
נהלי הקורס

- נא להיכנס לאתר הקורס:

<https://webcourse.cs.technion.ac.il/234123>

- ולקרוא את מסמך הנהלים תחת הלשונית "סילבוס".

מערכות הפעלה ...be like



Taken from: <https://www.flickr.com/photos/betbeder/14076789040/>

למה הקורס קשה?

- הesson חומר, המון קשרים בין הנושאים השונים.
- אין "גבولات גירה" מוגדרים, לא ניתן ל透פּוֹ את כל המערכת בבת- אחת.
- שיעור בית טכני בסביבת עבודה "קשה".
- אין debugger, עבודה בטרמינל, מכונה וירטואלית, קוד ספקטי עם goto, ...
- דורש הסתגלות לשפה חדשה: קורס הנדסי ולא מתמטי.

במערכות הפעלה	בחדי"א
שיפור של ϵ קטן לא מעניין	יהי $0 > \epsilon \dots$
תיאוריות המבוססות על תצפיות ועל היגיון	משפטים, אמת מוחלטת
מנסים למצוא פשרה בין מספר דרישות הנדסיות	מחפשים פתרונות אופטימליים

יש הרבה חומר כי גם בתרגולים לומדים חומר חדש (וגם קצת מתרגלים את החומר שכבר למדנו...).

אין גבולות גירה מוגדרים כי מאד קשה לבנות מערכת הפעלה שמיישמת את העקרונות של אבסטרקציה, מודולריות ואנקפסולציה כפי שלמדנו במת"מ.

למה בכלל זאת כדאי?

- אין ברירה – קורס חובה...
- של 4.5 נקודות!
- כי זה מעניין!
- לפחות אותו (;
- כדי לדעת איך עובדות מערכות מחשבים.
- מערכות מחשבים == המחשב הנייד שלכם, הפלאפון, וגם השרת של גугл שוננה לשאלות שלכם.
- כדי לעבור ראיונות עבודה באינטלק, מיקרוסופט, מלאנוקס, וכו'.
- מצד שני, אם רוצים להיות מפתח web, לא חייבים לדעת "מערכות הפעלה".

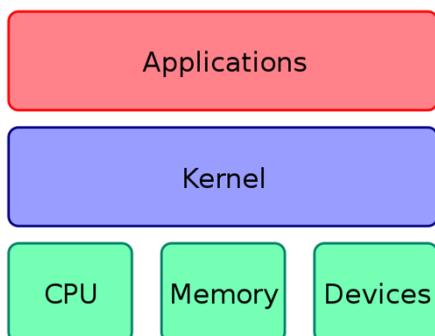
אוקי, השתכנעתי... איז איך מצליחים בקורס?

- אין חוכמות: משקיעים הרבה זמן ומשאצים.
- לומדים בצורה ספירלית.
- חוזרים לנושאים הקודמים בכל פעם שלומדים נושא חדש.
- לא מפחדים לשאול שאלות, לענות תשובות, ולטעות.
- חיוני כדי להתרגל לשפה החדשה.
- לא סובלים בגל שיעורי הבית.
- נתקעתם? תשאלו חברים או את המתרגל האחראי על התרגיל.
- להצלחה בלבד זה טוב; ללמידה אחרים זה עוד יותר טוב.

מהי מערכת הפעלה?

מהי מערכת הפעלה?

- מערכת תוכנה אשר אחראית על ניהול החומרה עבור המשתמשים.
- ליתר דיוק, זו ההגדרה של "גרעין" מערכת הפעלה.
- שלושת רכיבי החומרה המרכזיים הם:
(1) המעבד,
(2) הזיכרון,
(3) והדיסק.
- כל תרגול בקורס עוסק בניהול של אחד הרכיבים הללו.



"גרעין" מערכת הפעלה הוא החלק החשוב ביותר במערכת הפעלה, והוא לא כולל למשל את המשק הגרפי (שולחן עבודה עם אייקונים וחלונות).

בשქף מופיעה המילה התקנים (devices) של קלט/פלט, אבל בקורס נתמקד בעיקר בהתקן ספציפי, ואולי החשוב ביותר – דיסק קשיח.

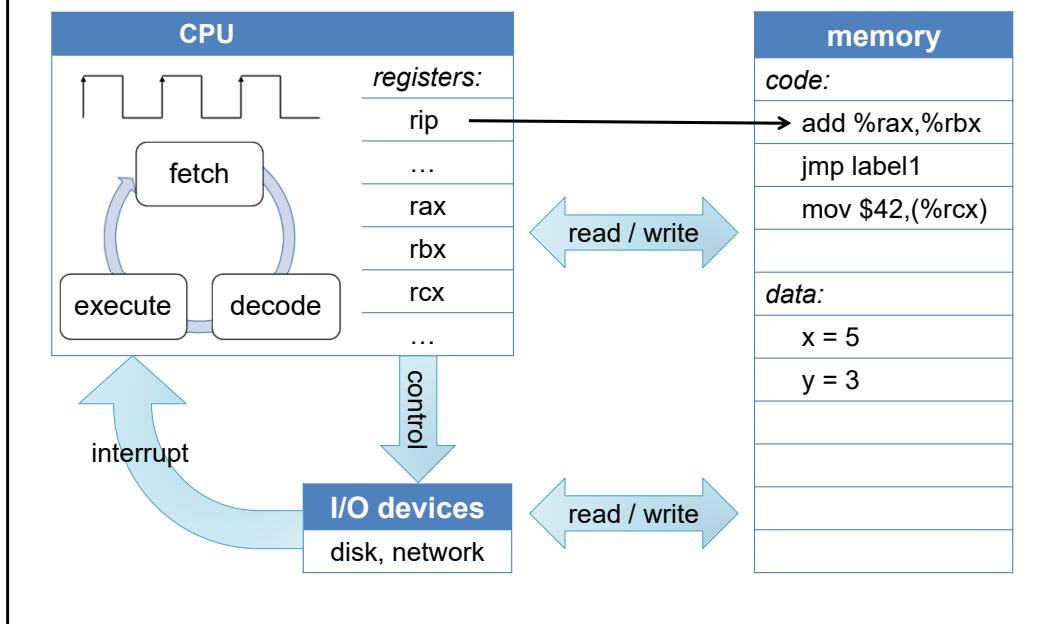
Picture from: [https://en.wikipedia.org/wiki/Kernel_\(operating_system\)](https://en.wikipedia.org/wiki/Kernel_(operating_system))

ישור קו

דיסק	זיכרון	معالג
амצעי אחסון איטי ועמד (המידע נשמר גם אם מכבים את המחשב)	амצעי אחסון מהיר ונדייף (המידע נמחק כאשר מכבים את המחשב)	מקבל זרם של פקודות מכונה (asmmbly) ובמציע אותן בצורה סדרתית
הمعالג לא יכול לעבוד מייד יישירות על הדיסק, הוא צריך קודם להעביר את המידע לזיכרון	הזיכרון נגיש לכל הליבות של אותו מחשב	כלمعالג מרכיב ממספר ליבות חישוב עצמאיות שרצות במקביל
נפח אופייני – 1TB זמן גישה טיפוס: ~1 ms	נפח אופייני – 8GB זמן גישה טיפוס: ~100 ns	תדר אופייני – 3GHz זמן גישה טיפוס: ~1 ns (רגיסטרים + מטמוןים)

בשქף זהה רצוי לעזר ולשאול את הסטודנטים מה הם יודעים עלمعالג, זיכרון, ודיסק לפני
שמציגים את הtablעה.

יחסים הקיימים בין רכיבי החומרה



תפקידו של מנגנון הפעלה

- לחלק את משאבי החומרה בצורה יעילה והוגנת בין המשתמשים.
- האתגר של מערכת הפעלה הוא לצורכי משאבי מועטים ככל הניתן.
- להציג למשתמשים אבסטרקציות וממשקים (API) כדי להקל את הפיתוח של אפליקציות.
 - אבסטרקציות לדוגמה: תהליך, מרחב זיכרון, קובץ, socket, ...
 - אוסף השירותים (== המשחק) שמערכת הפעלה מציגה נקרא **קריאות מערכת**.
- להגן על המידע של המשתמשים מפני משתמשים ו/או תוכניות אחרות זדוניות.
- לשמר על המידע של המשתמשים מפני נפילות חומרה.

תפקידים נוספים: חיסכון באנרגיה, ...

עקרון הוירטואלייזציה

הבעיה: חלוקת משאבים

- לIOS יש מעבד יחיד וזיכרון יחיד.
- אבל IOS רוצה להריץ הרבה הרובוט אפליקציות בו-זמנית.
- לגילוש באינטרנט, לשמעו מוזיקה, ולעבוד על שיעורי הבית במערכות הפעלה.
- הפתרון: ירטואלייזציה של משאבים פיזיים.
- מערכת הפעלה מציגה למשתמש גרסאות ירטואליות של המעבד והזיכרון.
- משאבים ירטואליים מספקים **אבסטרקציה** שמתעלמת מפרט המימוש של החומרה הספציפית, וכך הם פשוטים יותר לשימוש.
- משאבים ירטואליים גם מספקים **הגנה** כי הם מסתירים את המשאב הפיזי, וכן אף משתמש או אפליקציה לא יכולים להשתלט עליו.
- אבל ירטואלייזציה – כמו כל אבסטרקציה – גם מօיפה **תקורה** (overhead) שפוגעת ביצועים.

הבעיה של חלוקת המעבד תקפה גם אם יש כמה ליבות עיבוד (למשל 4), כי מספר התהליכיים במערכת יכול להיות גבוה יותר (למשל 40).

וירטואליזציה של המעבד

one physical CPU

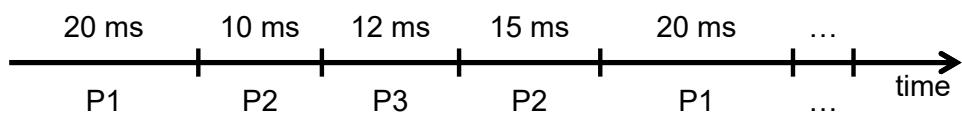


many (infinite) virtual CPUs

- אפליקציה רצה של יוסי תיקרא **תהליך**.
- מערכת ההפעלה מעניקה לכל תהליך את האשליה שיש לו מעבד וירטואלי נפרד ממשו.
- בcontra זו מערכת ההפעלה מקלה על התהיליך גם מגנה עליו.
- למשל, המעבד הוירטואלי מכיל את כל הרגיסטרים של המעבד הפיזי.
- התהיליך לא צריך "לחשב" באילו רגיסטרים מותר לו להשתמש כי המעבד הוירטואלי כלו שלו!
- כמו כן, תהליכיים אחרים לא יכולים לקרוא/לכתוב לרגיסטרים של התהיליך.

איך ממשיכים וירטואלייזציה של המעבד?

- מערכת הפעלה מחלקת את הזמן של המעבד הפיזי בין התהליכים השונים (time sharing).
- הגרעין מחליף במהירות בין התהליכים: מריץ תהליך אחד לפרק זמן קצר (מילישניות) ואז משנה את ביצועו וועובר להריץ תהליך אחר, וכן הלאה.
- המשמש מקלט אשליה של בו-זמניות אינטראקטיבית.



- פרטיים נוספים בתרגול בנושא החלפת הקשר.

בפועל, מערכת הפעלה מפעילה את הטכניקה המתוארת לעיל על כל **ליבת** של המעבד.

וירטואליזציה של הזיכרון

one physical
address space

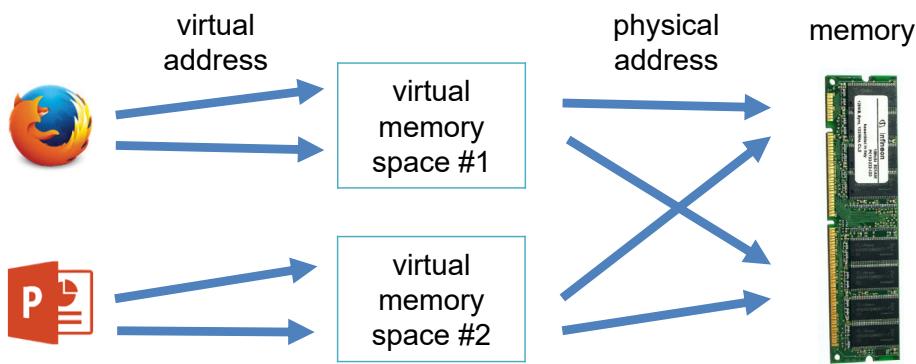


many virtual
address spaces

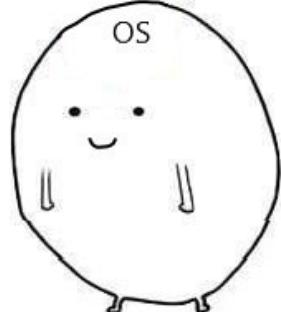
- מערכת ההפעלה מעניקה לכל תהליך את האשליה שיש לו מרווח זיכרון נפרד ממשו.
- בדומה זו מערכת ההפעלה מקלה על התהליך גם מגנה עליון.
- התהליך לא צריך "לחשב" באיזה זיכרון מותר לו להשתמש כי הזיכרון הוירטוואלי כלו שלו!
- כמו כן, תהליכי אחרים לא יכולים לקרוא/לכתוב לזיכרון של התהליך.

איך ממשיכים וירטואלייזציה של הזיכרון?

- מערכת ההפעלה והמעבד מתרגמים את הכתובות הווירטואליות לכתובות פיזיות.
- פרטים נוספים בתרגול בנושא זיכרון וירטואלי.

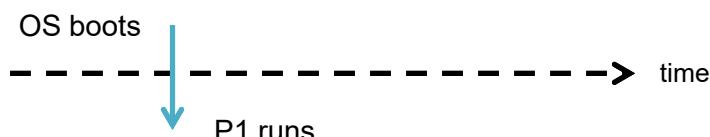


הפוך



אתגרים בימוש עיקרון הווירטואלייזציה

- **ניסיון ראשון:** הרצה ישירה (direct execution).
- נניח שיווי לחץ על כפטור הפעלה והדליק את המחשב.
- מערכת הפעלה היא הראשונה שרצתה.
- נתענת לזכרון, מזזה את רכיבי החומרה, מתחילה את מבני הנתונים שלה, ...
- לאחר שלב האתחול, מערכת הפעלה מעבירה את השליטה על המעבד הפיזי לתהיליך P1 כדי שירות על המעבד.



למה מערכת הפעלה מעבירה את המעבד לתהיליך?
הרצה ישירה על המעבד היא חיונית כדי להשיג ביצועים גבוהים לעומת אופציות אחרות,
לדוגמה אמולציה של המעבד, שבה מערכת הפעלה מתרגם את הקוד של התהיליך בזמן
ריצה ובודקת כל פקודה מוכנה שהטהיליך מבקש לבצע.

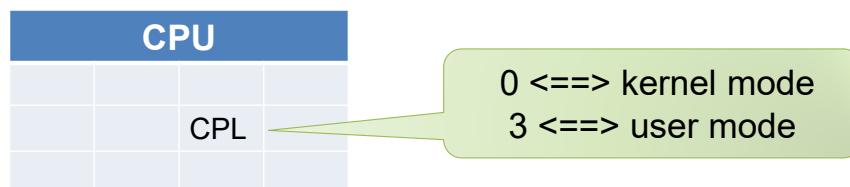
אבל מה אם...

- **בעיה #1:** תהיליך P1 ינסה לגשת לקבצים שאסור לו לגשת אליהם (למשל, קבצים של משתמש אחר במערכת)?
- **בעיה #2:** תהיליך P1 יריץ לולאה אינסופית וכך ימשיך להחזיק במעבד לנצח?
 - איך מערכת הפעלה תוכל להחזיר לעצמה את השליטה על המעבד?
 - למשל, כדי לעצור את תהיליך P1 ולהריץ במקוםו תהיליך אחר P2.
- **בעיה #3:** תהיליך P1 יבצע פעולה איטית מאוד (כמו קריאה/כתיבה מהדיסק) שלא רצה על המעבד?
- הפתרונותות לביעיות הללו מבוססים על מנגנוני חומרה שנלמד בערך.

This topic is summarized in Chapter 6 in OSTEP (“Mechanism: Limited Direct Execution”).

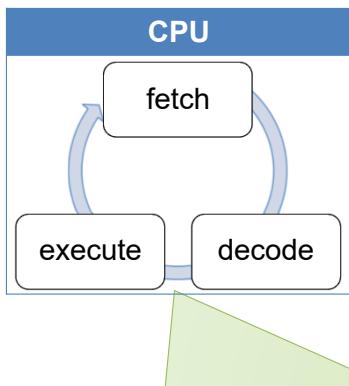
פתרון לבעה 1#: רמות הרשות

- המעבד יגדיר שתי רמות הרשות:
 - רמת הרשות נמוכה לשימוש תכניות רגילים (קוד משתמש). – **user mode**.
 - רמת הרשות גבוהה לשימוש מערכת הפעלה. – **kernel mode**.
- בכל רגע נתון, המעבד יהיה ברמת הרשות אחת ויחידה.
- רמת הרשות הנוכחית (CPL = current privilege level) נשמרת ברגיסטר **CPL** של המעבד.
- הפרטיהם המדוייקים נלמדים בקורס את"ם (234118).



שימוש לב: מספיק ביט אחד כדי לשמור את רמת הרשות הנוכחית, אבל במעבדי אינטל 64-ビיט מוקצים שני ביטים.

פקודות מיוחדות (privileged instructions)



- המעבד יחלק את פקודות המcona לשתי קבוצות:

 - פקודות מיוחדות (privileged).**
 - פקודות לא מיוחדות (non-privileged).**

- פקודות מcona שניגשות לדיסק, למשל, יהיו מיוחדות.

המעבד יבודק בשלב זה (לפני ביצוע הפקודה) האם היא מיוחדת. אם הפקודה מיוחדת, אבל המעבד במצב משתמש – תיווצר חריגה. החריגה תעביר את השליטה לגרעין, והוא ירוג את התהיליך.

יש לזכור כי בשלב זה לא כל הסטודנטים למדו מהי חריגה ומה המשמעות שלה (החומר נלמד בקורס אט"מ 234118).
 لكن כדאי להסביר בקצרה שחריגה היא שגיאה שנגרמה ע"י קוד המשתמש, כמו למשל חלוקה באפס או גישה לכתובת NULL.
 לא כל חריגה גורמת לסיום התהיליך שיצר אותה – יש חריגות שמסתיימות בשילוח סיגナル לתהיליך, והתהיליך יכול להתעלם מהסיגナル.
 אבל בשלב זה של הקורס (לפני שלמדנו על סיגנלים), נרצה לעצמנו להיות פחות מדויקים ;)

איך מושנים את רמת הרשאה?

- במילים פשוטות: איך משתמש ייגש לדיסק כדי לקרוא קובץ?

- ניסיונו ראשון: המשתמש יעלה את רמת הרשאה, ואז יקרא לפונקציה של מערכת הפעלה שתיגש לדיסק.

מה הבעיה
בחזעה זו?

- החזעה טובה יותר: המשתמש יקרא לפקודת מכונה מיוחדת אשר

מבצע שתי פעולות בעת ובעונה אחת:

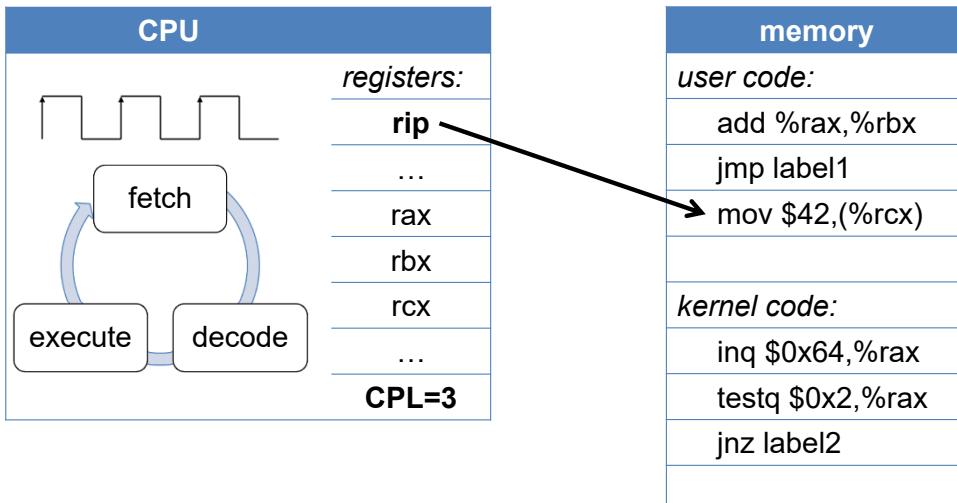
1. יעלה את רמת הרשאה.
2. תעבור לבצע פונקציה של מערכת הפעלה.

- הפקודה המיועדת להזמין **קריאה למערכת**, או באנגלית **System Call** (בקיצור).

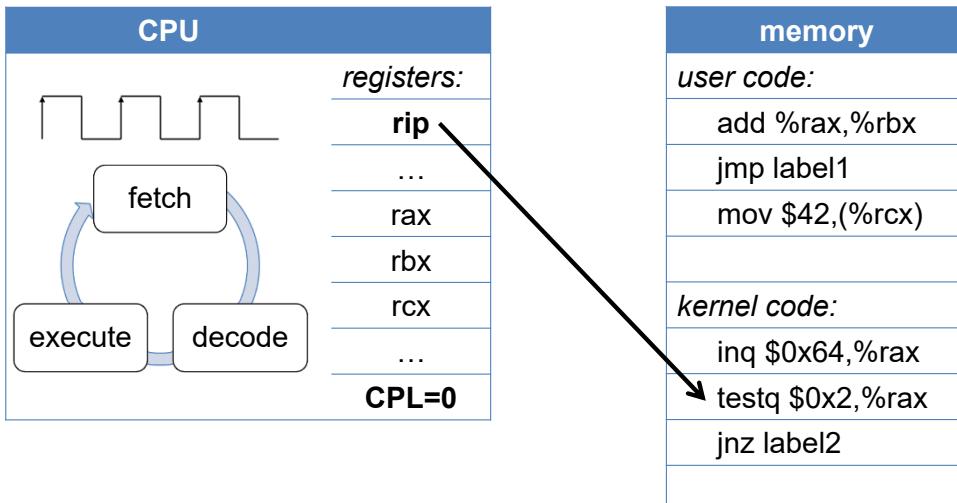
תשובה: אנחנו כמובן לא יכולים ולא רוצים לסמן על המשתמש שיעביר את השליטה למערכת הפעלה אחריו שהעלתה את הרשאות.

קריאה ל-call system שונה מקריאה לפונקציה "רגילה" – פרטיהם נוספים בקורס את "מ.(234118)

מצב המערכת לפני פקודה syscall



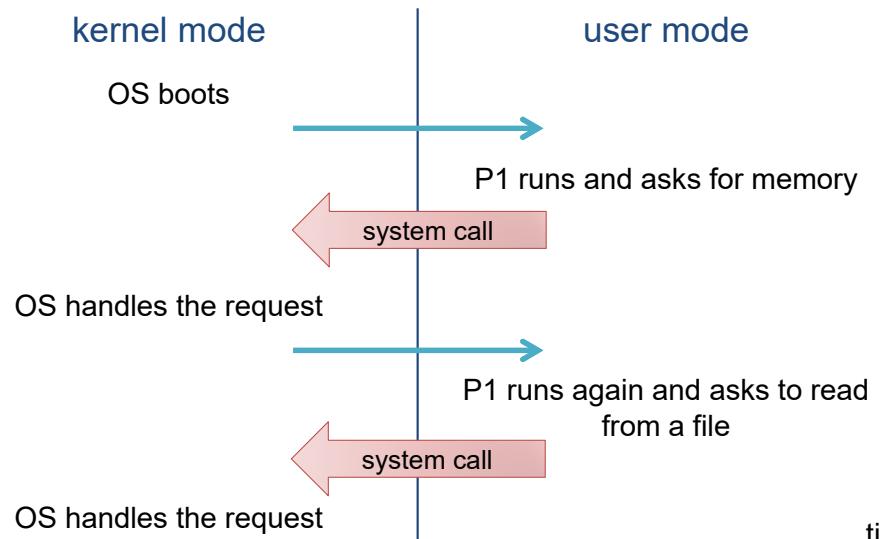
מצב המערכת אחרי פקודה syscall



קְרִיאֹת מִעֲרָכָה

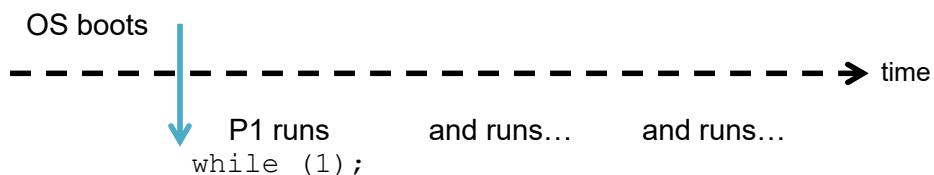
- קריאות מערכת מאפשרות העברת בטוחה וմבוקרת של השליטה על המעבד מפני שהן שער הכניסה **היחיד** של תוכניות משתמש לפקודות מיוחדות.
- קריאות מערכת מדירות למשזה את הממשק של מערכת הפעלה לתוכניות משתמש.
 - תהיליך משתמש קורא לקריאות מערכת כדי לבקש מערכת הפעלה שירות כלשהו כמו: לגשת להתקני קלט/פלט, ליצור תהילכים חדשים, לבקש עוד זיכרון ועוד.

תרחיש לדוגמה



זיכרון: בעיה #2

- מערכת הפעלה היא הראשונה שרצה לאחר הדלקת המחשב.
- בשלב מסוים, מערכת הפעלה מעבירה את השליטה על המעבד הפיזי לתהיליך P1 כדי Shirouz.
- אם P1 יקרא לקריאת מערכת, אז מערכת הפעלה תרוץ – ככלומר תקבל לידיה את השליטה על המעבד שוב.
- אבל אם P1 יריץ לולאה אינסופית (בטעות או בזדון)?
- הוא ימשיך להחזיק במעבד לנצח ולא יתאפשר תהיליך אחר לרוץ...



הפתרון היחיד לבעיה לעיל הוא לכבות את המחשב ולהדליק מחדש – פתרון גורע מאוד, כמובן.

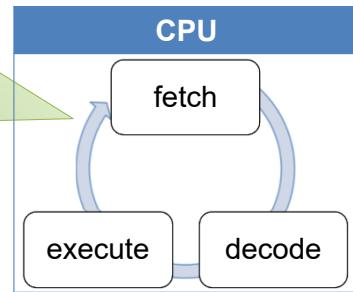
פתרון לבעה #2: פסיקות שעון

- לינוקס מפרקיעת (preempt) את המעבד מהתהיליך אחד לטובת תהיליך אחר, בעזרתו תקן חומרה מיוחד – **השעון** (timer).
- מערכת ההפעלה מבקשת מהשעון לשלוח פסיקה במרווחי זמן קבועים כדי להעביר את השליטה למערכת ההפעלה.
- כל הפסיקות, פרט לפסיקת שעון, מטופלות ב-mode kernel.
- במהלך הטיפול בפסיקה, מערכת ההפעלה יכולה להחליט שהוא מחליפה את התהיליך הנוכחי שרצץ כרגע על המעבד.
- הפעולה הזאת נקראת "**החלפת הקשר**" – נחזור אליה בהמשך הקורס.

משמעותו של השעון הוא רכיב חיוני למעבד והוא קשור לתדר השעון הפנימי של המעבד.
(זו טעות נפוצה של סטודנטים.)

הטיפול בפסיקות חומרה

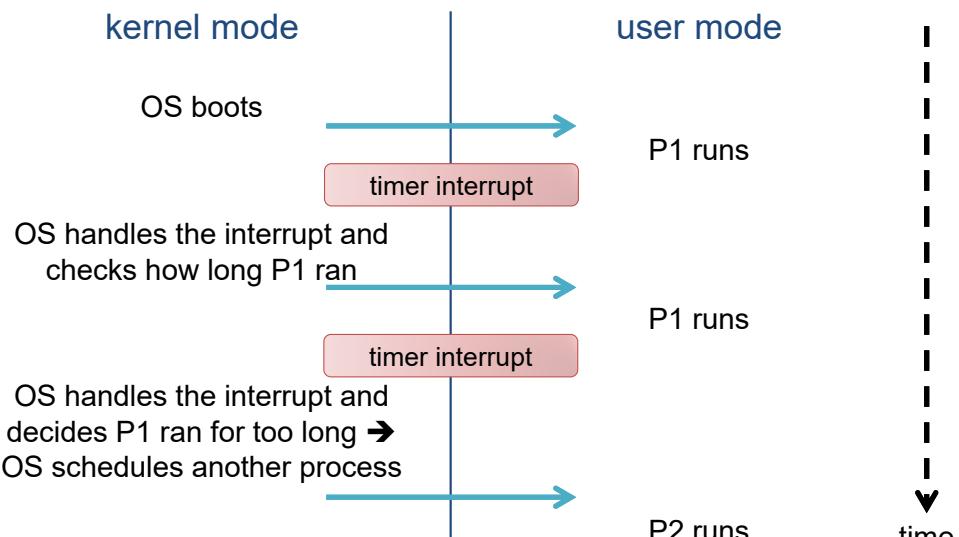
המעבד בודק אם יש פסיקות ממתיינות לאחר סיום של כל פקודת מכונה. אם יש פסיקה, המעבד מפסיק לבצע את הקוד הנוכחי וועבר לבצע את **שגרת הטיפול בפסיקה** (interrupt handler) במצב גרעין.



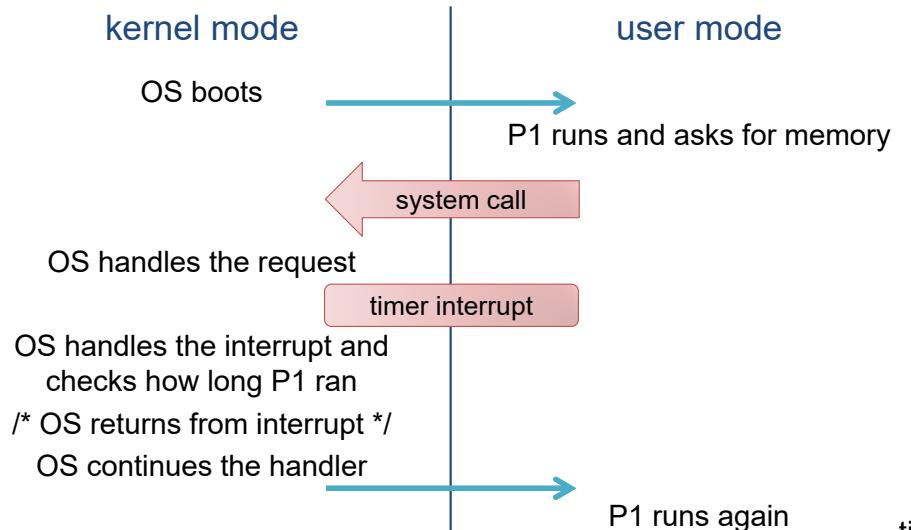
- לאחר סיום הטיפול בפסיקה, המעבד יחזיר לבצע את הקוד הקודם.
- כדי לדעת לחזור, המעבד ומערכת הרפעלה צריכים לשמור את המצב של המעבד ברגע קבלת הפסיקה – פרטיים נוספים בקורס אט"מ (234118).
- שימוש לב: פסיקה אינה קוטעת ביצוע של פקודת מכונה.
- פסיקות מטופלות "בין" פקודות מכונה.

בין אם הוא במצב משתמש או מצב גרעין

תרחיש לדוגמה



פסיקות יכולות להגיע גם במצב גרעין !

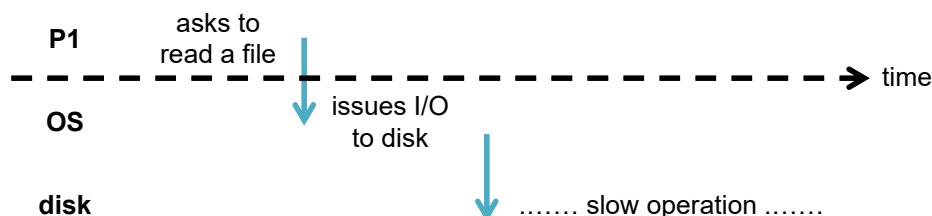


שאלה לסטודנטים: ראיינו שפסיקה יכולה לקטוע קריית מערכת. האם קריית מערכת יכולה לקטוע פסיקה?

תשובה: לא, קריית מערכת לעולם לא תגיע בזמן טיפול בפסיקה.
קריית מערכת היא בקשת שירות של המשתמש ממערכת הפעלה.
אבל פסיקות מטופלות במצב גרעין ולכן לא צרכות לבקש (וגם לא מבקשת) שירותים ממערכת הפעלה.

זיכרון: בעיה #3

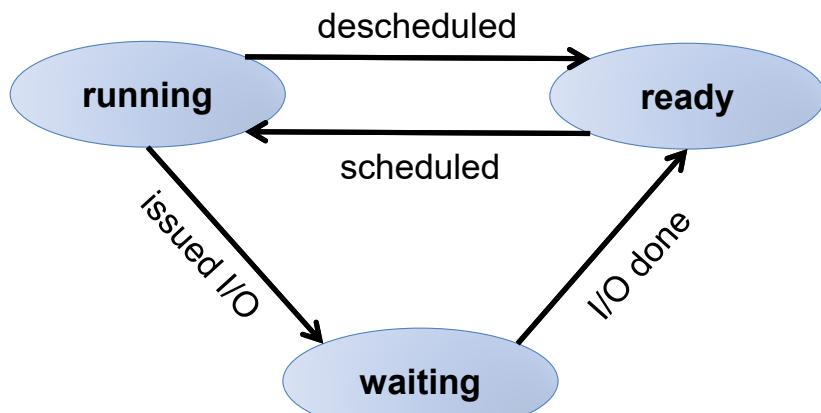
- תהלייר יכול לבקש מערכות הפעלה שירות O/I, לדוגמה:
קריאה/כתיבה מהדיסק או מכרטיס הרשות.
- גישה להתקני O/I היא איטית מאוד: סדר גודל של מספר מילישניות
= מיליאני פקודות מעבד.
- בזמן הבדיקה להתקני O/I התהלייר לא רץ והמעבד חסר פעילות.
- איך נוכל לנצל טוב יותר את המשאבים של המערכת?



נרצה שגם המעבד וגם הדיסק יהיו עוסקים בו-זמנית כדי לקבל נצילות גבוהה של המערכת.

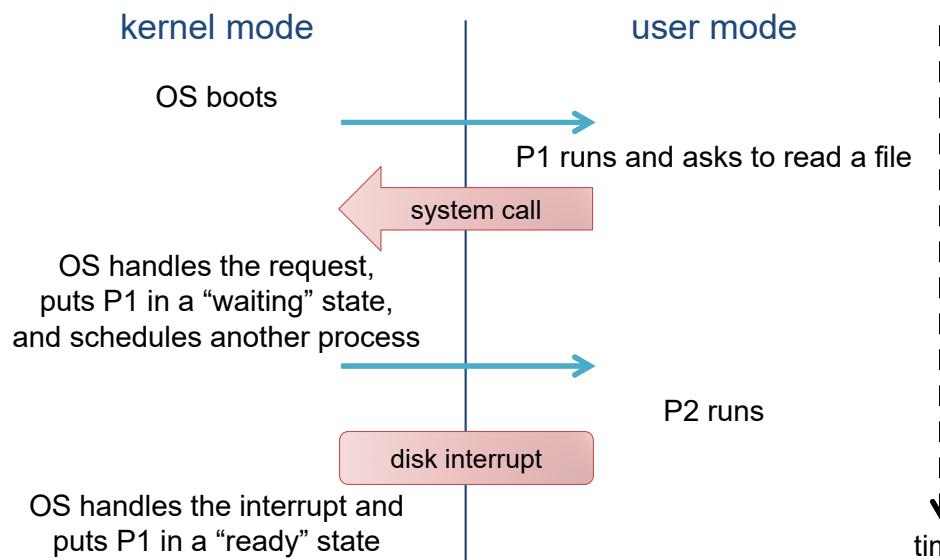
פתרון לבעה #3: מצבים המתנה

- מערכת הפעלה תסואג את התהילcis במערכת לשלושה מצבים אפשריים (בפועל יש יותר, אלו המצבים העיקריים):



The “Waiting” state is sometimes called “Blocked” because the process cannot progress in this state.

תרחיש לדוגמה



LOSECUM

- בעיות #1 , #2 הן בעיות אבטחה (security).
- בעיה #3 היא בעית יעילות (efficiency).
- מערכת ההפעלה שואפת להשיג גם יעילות וגם אבטחה, ולכן היא פותרת את שלושת הבעיה הללו באמצעות מגננון:

הרצה ישירה מוגבלת (limited direct execution)

אבל כדי לשמור על אבטחה, התהליכים לא יכולים להריץ כל פקודה.

כדי לקבל ביצועים גבוהים, התהליכים רצים ישירות על המעבד הפיזי.

מערכת הפעלה "لينוקס"

קצת היסטוריה...

-
- The timeline consists of five blue chevron-shaped boxes arranged vertically, each containing a historical event and its year. The years are 1973, 1983, 1990, 1991, and 1992.
- דניס ריצ'י וקן תומפסון ממעבדות Bell מפתחים מערכת הפעלה קניינית בשם יוניקס (UNIX). **1973**
 - ריצ'רד סטולמן מכריז על מיזם GNU במטרה לפתח מערכת הפעלה חופשית תואמת יוניקס (וגם ספריות וכלים נוספים). **1983**
 - מיזם גנו מתחילה לפתח את גרעין מערכת הפעלה, אך הפיתוח מתגלה כמסובך ומתකדם באטיות רבה. **1990**
 - לינוס טורබאלד מתחילה לפתח את גרעין לינוקס במהלך לימודיו באוניברסיטת הלסינקי. **1991**
 - טורבעאלד משנה את רישיון לינוקס ל-[GPL](#) וכן לינוקס הופכת לגרעין מערכת הפעלה של מיזם GNU. **1992**

שאלת: מה עוד המציא דניס ריצ'י?
תשובה: שפת C. השפה הזאת אפשרה לינוקס (שנכתבה בשפת C) לעבור בקלות בין ארכיטקטורות מעבדים שונות, בניגוד למערכות הפעלה קודמות שנכתבו בשפת אסמבלי.

בקורס נלמד ונשתמש בלינוקס

- הסיבה המרכזית לכך: לינוקס היא תוכנה חופשית וקוד פתוח.
- קוד המקור של גרעין לינוקס זמין לשימוש, לשינוי ולהפצה בחינם לכל אחד.
- היום השם "لينوكס" מתייחס למשפחה של מערכות הפעלה המבוססות על גרעין לינוקס ורכיבי התוכנה של פרויקט GNU.
- ההפעלה הספציפית בה נשתמש בקורס היא Ubuntu 18 LTS.
- גרסת הגרעין היא 4.15.
 - כל הקוד המקורי חופשי לכולם באינטרנט:
<https://elixir.bootlin.com/linux/v4.15.18/source>.
- בתרגילים בית 0 תתקינו את מערכת ההפעלה לינוקס על המחשב האישי שלכם באמצעות מכונה וירטואלית (virtual machine).

בקורס בחרנו לעבוד עם שולחן העבודה XFCE, שהוא לא ברירת המחדל בהפעלה של אובונטו, אבל הוא רזה יותר ולכן קל יותר להריצה גם על מחשבים חלשים.

נא לרשון את החומר של מת"מ (234124)

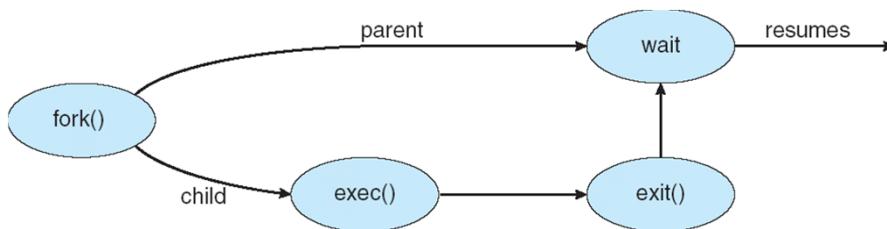
- עבודה ב-shell .bash
- פקודות בסיסיות: ...cd, ls, date
- הרצה בחזית או ברקע.
- הפנימית קלט/פלט, pipes.
- שלבי בניה תוכניות: קומpileציה ו קישור.
- אוטומציה של בניה תוכניות באמצעות Makefile.

תרגול 2

קריאות מערכת לעבודה עם תהליכיים
ניהול תהליכיים בגרעין לינוקס
תורי תהליכיים

TL;DR

- תכנית היא אוסף פקודות; תהליך הוא ביצוע של אותן פקודות.
- בשעה הראשונה: נלמד איך **קוד משתמש** יכול ליצור תהליכי חדשים, לברר מה מצבם, ולהמתין לסיום שלהם.
- באמצעות קרייאות המערכת: **fork**, **execv**, **exit**, **wait** (ועוד כמה).
- המשך לא איןטואיטיבי במבט ראשון.



- בשעה השנייה: נלמד איך **הגרעין** מմמש את קרייאות המערכת הללו.

From: <http://www.cs.rpi.edu/~goldsd/docs/fall2014-csci4210/04-fork-diagram.png>

קריאות מערכת לעבודה עם תהליכיים

מהו תהליך?

- תהליך (**process**) הוא ביצוע סדרתי של תכנית (**program**).
 - תהליך = מופע (**instance**) של ביצוע תכנית.
- מערכת הפעלה נותנת לכל תהליך אשליה שהוא לבד במערכת כדי להקל על פיתוח אפליקציות ו כדי לספק לתהליכיים הגנה זהה מזה.
- אבל תהליכיים יכולים גם לתקשר ביניהם – נלמד בהמשך הקורס.
- מספר תהליכיים רצים "בזמנית" על המעבד: מערכת הפעלה מחליפה בין התהליכיים במתירויות ויוצרת אשליה שהם רצים יחד.
 - בהמשך הקורס נלמד איך לנוקס ממשת את ההחלפה בין(processes).
- כל תהליך צריך משאבים, למשל: זמן מעבד, זיכרון, ...
 - בהמשך הקורס נלמד איך לנוקס מחלוקת זמן המעבד בין-processes.

תהליך נקרא גם: משימה, task או job .

תהליכיים בסביבה Windows

The screenshot shows the Windows Task Manager interface. At the top, there are tabs: מנהל המשימות (任务管理器), קבץ אפשרויות תצוגה (显示设置), היסטוריית אפליקציות (应用历史记录), משתמשים (用户), פרטם (详细信息), שירותים (服务), ביצועים (性能), and תהליכיים (任务). The 'תהליכיים' tab is selected.

The main area displays a table of processes. The columns are: רשת (Network), 0% (0%), 1% (1%), זיכרון (Memory), 56% (56%), 4% (4%), CPU, PID, שם (Name), and additional columns for application icons and names.

Two sections are highlighted:

- אפליקציות (5)**: Contains five entries: Firefox (3 instances), Microsoft PowerPoint, Microsoft Word, מנהל המשימות (Task Manager), and סייר (File Explorer).
- תהליכיים ברקע (78)**: Contains three entries: Adobe Acrobat Update Service, Antimalware Service Executable, and Application Frame Host.

At the bottom left is a 'סימן משימה' (Pin Task) button, and at the bottom right is a 'פחתן פרטם' (Minimize Details) button.

רשת	0%	1%	זיכרון	56%	4%	CPU	PID	שם	
0 Mbps	0.1 MB לשניה	294.3 MB	1.3%					(3) Firefox	<
0 Mbps	0 MB לשניה	75.4 MB	0%				7096	Microsoft PowerPoint	<
0 Mbps	0 MB לשניה	44.5 MB	0%				12952	Microsoft Word	<
0 Mbps	0 MB לשניה	24.5 MB	0.2%				12872	מנהל המשימות	<
0 Mbps	0.1 MB לשניה	36.8 MB	0.3%				13988	סייר	<
									אפליקציות (5)
0 Mbps	0 MB לשניה	1.0 MB	0%				4604	...32) Adobe Acrobat Update Service	<
0 Mbps	0.1 MB לשניה	106.8 MB	0.3%				11284	Antimalware Service Executable	<
0 Mbps	0 MB לשניה	3.9 MB	0%				9768	Application Frame Host	<
									תהליכיים ברקע (78)

תהליכיים בלינוקס

- לכל תהליך בלינוקס יש מזהה הקורי PID – Process IDentifier.
- מספר שלם בן 32 ביט, ייחודי לתהליך.
- ברוב מערכות לינוקס משתמשים רק ב-15 הביטים התחכוניים, ולכן ניתן ליצור עד 32K תהליכיים. מנהל המערכת יכול להגדיר מספר גבוי יותר של תהליכיים.
- שימוש לב: ערכי ה-pid ממוחזרים מתחלכים שסימנו לתהליכיים חדשים.
- עם עליית המערכת, הגרעין יוצר את התהליך idle שמספרו 0=pid.
 - נקרא לריצה כאשר אין תהליכיים מוכנים לריצה ומבצע פקודת מכונה hlt, המכנישה את המעבד למצבamina.
- התהליך idle יוצר את התהליך init שמספרו 1=pid.
- התהליך init ייצור את כל שאר התהליכיים.

למה זה כדאי?

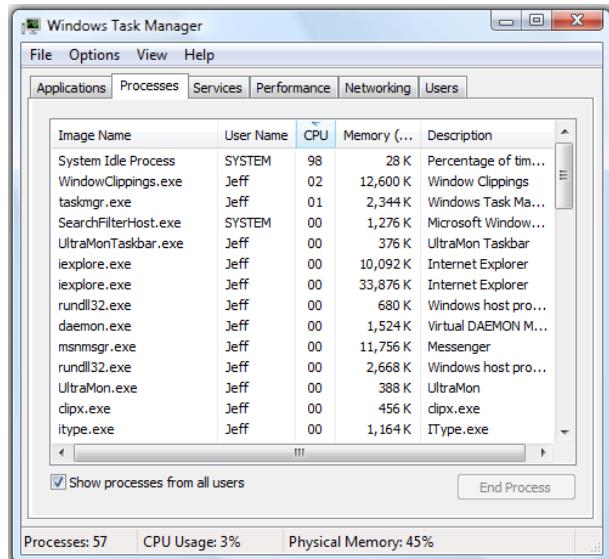
Answer: To save power.

It is important to note that idle is not very “idle” but integrates many of the maintenance kernel threads under its umbrella.

In the [x86 computer architecture](#), HLT (halt) is an [assembly language](#) instruction which halts the [central processing unit](#) (CPU) until the next external [interrupt](#) is fired.^[1] Interrupts are signals sent by hardware devices to the CPU alerting it that an event occurred to which it should react. For example, hardware timers send interrupts to the CPU at regular intervals.

The HLT instruction is executed by the [operating system](#) when there is no immediate work to be done, and the system enters its [idle state](#). In [Windows NT](#), for example, this instruction is run in the "[System Idle Process](#)". On x86 processors, the [opcode](#) of HLT is 0xF4.

אפל בסביבה Windows



קריאה המערכת (`fork()`)

```
pid_t fork();
```

- **פעולה:** מעתקה את תהליך האב לתהליך הבן ו חוזרת בשני התהליכיים.

- **קוד זהה** (ומייקום בקוד).
- **דיכרין זהה** (משתנים וערכיהם, גם במחסנית וגם בערימה).
- **סביבה זהה** (קבצים פתוחים, ספריית עבודה נוכחת).
- אבל, תהליך הבן הוא תהליך נפרד מתהליך האב, לכן יש לו **PID משלה**.
- **פרמטרים:** אין.
- **ערך מוחזר:**

 - במקרה של כישלון: 1 - לאב (אין בן).
 - במקרה של הצלחה: לבן מוחזר **0** ולאב מוחזר ה-**pid** של הבן.

אי אפשר להבדיל בין אב
לבן אם ה-pid של הבן
ויצא במקרה 0?

במקרה של הצלחה תמיד יוחזרו ערכים שונים לאב ולבן, כי ה-PID של הבן שונה מ-0
(הזהה הזה תפוא ע"י idle).

לפני fork()

parent

```
int main() {  
    int x = 0;  
    → pid_t p = fork();  
    if (p == 0) {  
        x = 1;  
    } else {  
        x = 2;  
    }  
}
```

אחרי fork()

parent

```
int main() {  
    int x = 0;  
    pid_t p = fork();  
    → if (p == 0) {  
        x = 1;  
    } else {  
        x = 2;  
    }  
}
```

son

```
int main() {  
    int x = 0;  
    pid_t p = fork();  
    → if (p == 0) {  
        x = 1;  
    } else {  
        x = 2;  
    }  
}
```

קוד האב וקוד הבן מסתעפים

parent

```
int main() {
    int x = 0;
    pid_t p = fork();
    if (p == 0) {
        x = 1;
    } else {
        → x = 2;
    }
}
```

son

```
int main() {
    int x = 0;
    pid_t p = fork();
    if (p == 0) {
        → x = 1;
    } else {
        x = 2;
    }
}
```

מה יהיה ערכו של x
בסוף דבר?
1 או 2?

תשובה: השאלה נועדה לבבל, כי שתי התשובות נכונות.
לכל אחד מהתהליכים עותק נפרד של x, ולכן ערך אחר של x.

שכפול מרחב הזיכרון ע"י `fork()`

- לאחר פעולה `(fork()` מוצלחת, אמן יש לאב ולבן את אותם משתנים בזיכרון, אך **בעותקים נפרדים**.
- כלומר, שינוי ערכי המשתנים אצל האב לא ייראה אצל הבן, וההיפך.

הדפסה לא מתואמת למסך

- אם ()fork נכשלה:
hello
 - שאלת: מה מძפיס הקוד הבא אם fork נכשלת? ואם היא מצליחה?
 - ואם ()fork הצלחה?
 - יש הרבה תשובות אפשריות, למשל:
hellohello
hheellollo
helhellolo
- הסיבה: שני התהליכיים ניגשים彼此ה לא מתואמת **למסך** משותף - המסך.

```
int main() {  
    fork();  
    printf("hello");  
    return 0;  
}
```

Advanced Note: today, printf() is thread safe, i.e., a lock protects the internal STDOUT from corruption and promises that no thread/process interrupts another when it is pushing strings to the buffer. Therefore, only hellohello is a possible outcome.

קריאה המערכת (wait())

```
pid_t wait(int *wstatus);
```

- **פעולה:** ממתינה עד אשר אחד מתהליכי הבן יסימם.

- **פרמטרים:**

- wstatus – מצביע למשתנה בו יאוחסנו פרטיים על תהליך הבן שהסתיים.

- למשל, wsstatus יכול את ערך הסיום של הבן (הערך שהעביר כארגומנט ל-exit()). ערך הסיום מופיע בבית השני מתוך ארבעת בתיה-wstatus. כדי לחילץ אותו יש לנצל את המאקרו (WEXITSTATUS(&wsstatus) & 0xff).

- במידה ולא מעוניינים בסטטוס הבן שסימם, אפשר להעביר NULL.

מבדים את החישוב?

ערך מוחזר:

- אם אין בניים או של הבנים כבר סיממו ובוצע להם (wait) – יוחזר מיד הערך -1.
- אם יש בניים שסיממו ועודין לא בוצע עבורם (wait) (כלומר הם במצב zombie – יפורט בשיקופיות הבאות) – יוחזר מיד ה-pid של אחד הבנים הנ"ל.

אחרת – המתנה עד שבן כלשהו יסימם.

- לפיזם כל תהליכי הבן?

תשובה:

```
while (wait(NULL) != -1);
```

קריאה המערכת `wait` אחורית על שחרור המשאב שהוקצה בקריאה שלם המערכת `fork`.

נוזח לזכור את ההקבלה: `fork ⇔ malloc`, `wait ⇔ free`

הדפסה מתואמת למסך

- שימוש ב-`-fwait` יכול לפטור את הבעיה שראינו קודם קודם כאשר מדפיסים למסך במקביל משני תהליכיים:

```
int main() {  
    pid_t p = fork();  
    if (p > 0) {  
        // parent waits for child  
        wait(NULL);  
    }  
    printf("hello");  
    return 0;  
}
```

קריאה המערכת `waitpid()`

```
pid_t waitpid(pid_t pid, int *wstatus,  
                int options);
```

- פעולה: המתנה לסיום בן **מדויק** שמספרו pid.
- ()((), wait(), waitpid() קריאות מערכת חוסמת.
- כלומר חוסמת את התקדמות התהיליך עד להתרחשות תנאי מסויים.
- **באנגלית**: **blocking system calls**
- הארגומנט options מאפשר לשנות את ההתנהגות של ()(pid של waitpid לקריאה מערכת לא חוסמת.
- אם **G options==WNOHANG** קריאת המערכת תחזיר מיד, כאשר ערך חזרה 0 משמעוינו שאף תהיליך בן עד לא סיים, ואילו ערך חזרה חיובי הוא ה-pid של תהיליך בן שסיים ונמצא עדין במצב **zombie**.

קריאה המערכת (exit())

```
void exit(int status);
```

- פעולה: מסיימת את ביצוע התהיליך הקורא ומשחררת את כל המשאבים שברשותו. התהיליך עובר למצוב **zombie** עד שתהיליך האב יבקש לבדוק את סיוםו ואז יפונה לחלווטין.

מה המטרת של
מצוב זה?

פרמטרים:

- `status` – ערך סיום המוחזר לאב אם יבודק את סיום התהיליך.
- בפועל ניתן להעביר להורה רק 8 ביטים בתווך ערך סיום, ולכן קריית המערכת תעבור `(status & 0xff)`.
- ערך מוחזר: הקריאה אינה חוזרת.
- לפי ה-`ח-מן`, קריית המערכת `exit` לא יכולה להימשך.

Answer: Enable the father process the option to check-up on the son's exit reasons and internal information.

קריאה המערכת (exit)

- שאלה: למה בכלל לקרוא ל-(`exit(status)` , אם אפשר פשוט לרשום `return status` בסוף פונקציית `main`?

- תשובה: `main` היא לא אמת הפונקציה הראשית של התוכנית....

- `main()` נקראת ע"י `libc_start_main` שاؤוספת את ערך החזרה של `main` וקוראת ל-(`exit()`).

```
int __libc_start_main(...){  
    ....  
    exit(main(...));  
}
```

- מסקנה: הפונקציה `exit` תמיד נקראת לסיום סטנדרטי של התוכנית.

לכארה, הסטנדרטו עד היום גם בלי קראת המערכת (`exit` ופשוט סיימנו תהליכיים באמצעות `return` בסוף פונקציית `main`).
 לכן חשוב להציג לסטודנטים ש-`return` היא לא קסם, וחיבים את עזרת מערכת הפעלה (באמצעות קראת המערכת `exit`) כדי לסייע תחילר.
 למה חיבים את עזרת מערכת הפעלה? למשל כדי להעיר את תחילר האב שחייב להתחיל שזה עתה מסתיים באמצעות `exit`.

הערה: יש עוד סיבה לשימוש ב-(`exit`), והוא נוחות למפתח התוכנה.
 (`exit` מאפשרת לקוד שנמצא עמוק בתוך שרשרת קראות לפונקציות לסיים את התחילר באופן מיידי במקום להחזיר ערך שגיאה אחרת עד לפונקציית `main`).

סיום תהליכיים

- כדי לאפשר לאב לקבל מידע על סיום הבן, לאחר שתהליך מסיים את פעולתו הוא עובר למצב מיוחד – **zombie** – שבו התהליך קיימם כרשומת נתונים בלבד ללא שום ביצוע ממשימה.
- הרשומה נמחקת לאחר שהאב קיבל את המידע על סיום הבן באמצעות(`wait()`).
- **שאליה:** מה קורה לתהליך "יתום" (orphan), ככלומר תהליך שישים לאחר שאביו כבר סיים בלי לקרוא ל-`wait()` ?
 - התהליך הופך להיות בן של `init`.
- התהליך `init` ממשיר להתקיים לאורך כל פעולת המערכת.
- אחד מתפקידיו העיקריים – המתנה לכל בניו כדי לפנות את נתוניםם לאחר סיוםם.

בהרצאות קוראים למצב זומבי בשם `terminated`.
 הפונקציה שאחראית על שינוי האבא לתהליך יתום היא (`forget_original_parent()`) והוא נקראת במהלך קריית המערכת (`exit()`) **של האבא**. (קובץ `gruen.c`)

קריאה המערכת (execv)

```
int execv(const char *filename,  
          char *const argv[]);
```

- פעולה: טוענת תוכנית חדשה לביצוע במקום התהיליך הקורא.

- פרמטרים:

- filename – מסלול אל הקובץ המכיל את התוכנית לטעינה.
- argv – מערך מצביעים למחוזות המכיל את הפרמטרים עבור התוכנית. האיבר הראשון מקיים `filename == argv[0]`, דהיינו מכיל את שם קובץ התוכנית. האיבר האחרון הפרמטר האחרון מכיל `NULL`.

- ערך מוחזר:

- במקרה של כישלון: `-1`.
- במקרה של הצלחה: הקריאה אינה חוזרת. איזורי הזיכרון (קוד, מחסנית, ...) של התהיליך מאותחלים עבור התוכנית החדשה שמתחילת להתבצע מההתחלה.

למה צריך NULL?

Answer: execv() can receive an arbitrary number of parameters, so we need to specify to the program where is the last one.

Note: execv() is actually one of a large family of execute functions. You can read more in the **man** pages:

<https://linux.die.net/man/3/execv>

קריאה המערכת (execv)

- מה ידפס הקוד הבא?

```
int main() {  
    char *argv[] = {"date", NULL};  
    execv("/bin/date", argv);  
    printf("hello");  
    return 0;  
}
```

- התשובה:

- אם execv מצליחה: את התאריך והשעה.
- אם execv נכשלת: hello

קריאה למערכת getpid(), getppid()

```
pid_t getpid();
```

- קריאה למערכת המחזירה לתהיליך הקורא את ה-pid שלו עצמו.

```
pid_t getppid();
```

- קריאה למערכת המחזירה את ה-PID של תהיליך האב של התהיליך הקורא.

- שאלה:** מה המשמעות של `1 == (pid == getppid())` עבור תהיליך משתמש?

- תשובה:** תהיליך האב הוא זה. קורא למשל אם תהיליך הבן יתום.

סיכום: עבודה עם תהליכי בילוקס

- תהליך חדש יכול להיווצר אך ורק ע"י **העתקה** של תהליך קיימ, באמצעות קריית המערכת (`fork()`).
- התהליך המקורי נקרא תהליך אב (או הורה), התהליך החדש נקרא תהליך בן.
- העותק של תהליך הבן זהה לגמרי **פרט למספרת התהליך (PID)**.
- תהליך אב יכול ליצור יותר מתהליך בן אחד.
- לאחר הייצורו, תהליך הבן יכול לבצע משימה שונה מאביו **על-ידי** הסתעפות בקוד התכנית בהתאם לערך החזרה של (`fork()`).
- תהליך הבן יכול לטען תכנית חדשה לביצוע **על-ידי** קריית המערכת (`execv()`).
- תהליך אב יכול לבדוק או להמתין **לסיום** תהליך בן שלו **על-ידי** קריית המערכת (`wait()`).
- אב יכול לבדוק סיום של בניים שלו, **אך לא של "נכדים", "אחים" וכדומה**.

דוגמת קוד מסכמת**פלט לדוגמה:**

```
pid = 8919
child pid = 8920
Sun Oct 29
00:31:32 IDT 2017
parent pid = 8919
```

```
printf("pid = %d\n", getpid());
pid_t pid = fork();
if (pid == 0) {
    printf("child pid = %d\n", getpid());
    char* args[] = {"./bin/date", NULL};
    execv(args[0], args);
    printf("This should not be printed\n");
} else {
    wait(NULL);
    printf("parent pid = %d\n", getpid());
}
```

Pids are made sequentially: 0,1,2,3....

headers:

```
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>
```

אתחול תהליכי בלינוקס

- משתמשים מתחברים לрабודה בלינוקס דרך מסופים (terminal).
 - מסוף = מסך + מקלדת (מקומי או מרוחק).
- התהיליך יוזם יוצר תהליך בן עבור כל מסוף, אשר טוען ומבצע את המשימות הבאות לפי הסדר:
 - .1. אתחול של המסוף.
 - .2. התחברות של המשתמש עם שם משתמש וסיסמה באמצעות תכנית `login`.
 - .3. אם אושרה כניסה המשתמש: קריאה לתוכנית **shell** (כמו `tcsh` או `bash`) המאפשרת למשתמש להעביר פקודות למערכת הפעלה.

דוגמה לשימוש בתהליכי shell

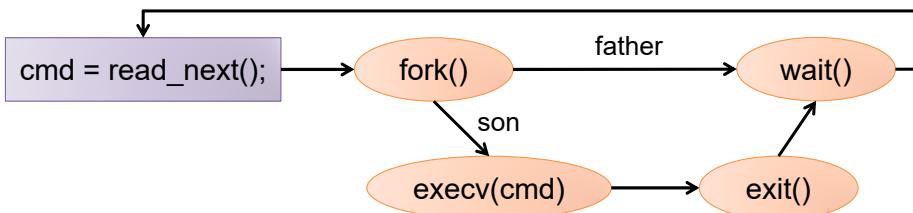
- ממשך שורת פקודה (command line).
- "יעוד עיקרי": לקבל פקודות ולבצע אותן באופן סדרתי.
- ה-shell מייצר תהליך בן עבור כל פקודה על-מנת לבצע אותה.
- כל פקודה ניתן להריץ בחזית (foreground) או ברקע (background).
- הריצה בחזית: האב (shell) ממතין לסיום הבן לפני קריית הפקודה הבאה.
- הריצה ברקע: האב (shell) עובר מיד לקריית הפקודה הבאה.
- "יעוד נוסף": להציג קבצים ותיקיות על-מנת לסייע במערכת.
- דוגמה חייה:
https://www.tutorialspoint.com/unix_terminal_online.php .

Open the online terminal and demonstrate the difference between:

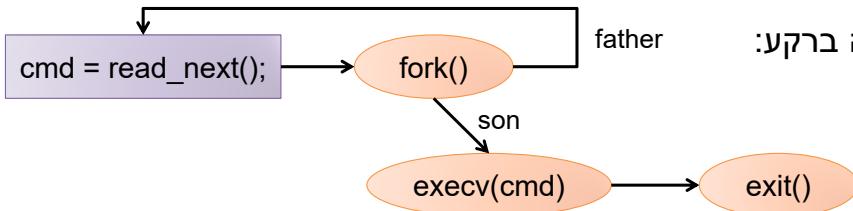
>> sleep 20
>> sleep 20 &

אופן פעולה shell בLinux

- הרצה בחרזית:



- הרצת ברקע:



Note that when running in the **background**, the shell **eventually** does wait on the user processes, only for the purpose of cleaning their records from the OS tables.

1

שאלה מבחן

.1. נתונות שתי תוכניות:

```
int main() {
    if (fork() > 0) {
        wait(NULL);
    } else {
        printf("I'm Pickle Rick");
    }
    return 0;
}
```

A

```
int main() {
    if (fork() > 0) {
        // do nothing
    } else {
        printf("I'm Pickle Rick");
    }
    return 0;
}
```

B

- מרכיבים את תוכנית A בלוולאה בעזרת הפקודה הבאה ב-shell :
- ```
for i in {1..N}; do ./prog; done
```
- שאלה: כמה תהליכי **כל** יותר יכולים להתקיים במערכת ברגע כלשהו?
  - ללא התחשבות בתהיליך shell עצמו או תהליכי אחרים שאינם נתונים בשאלה.

א. 1      ב. 2      ג. N      ד. N+1      ה. N+2      ו. 2N

- ומה התשובה אם מרכיבים את תוכנית B בלוולאה?

For A: The father waits for the son, and therefore, each iteration of the bash loop cannot end until both the son and father have finished working. Therefore, a maximum of 2 processes can run on the system, excluding the Shell process.

For B: It is possible that the father would finish **before** the son, thus allowing for another iteration of the bash loop. At the worst case, a total of  $N$  children would be running, and an additional father that has not yet finished running. We would have  $N+1$  processes in that scenario, running concurrently.

.2. נתונות אותן שתי תוכניות:

```
int main() {
 if (fork() > 0) {
 wait(NULL);
 } else {
 printf("I'm Pickle Rick");
 }
 return 0;
}
```

A

```
int main() {
 if (fork() > 0) {
 // do nothing
 } else {
 printf("I'm Pickle Rick");
 }
 return 0;
}
```

B

- כתעת מרכיבים כל איטרציה של הלולאה בפרק :

`for i in {1..N}; do ./prog &; done`

- שאלה: כמה תהליכי **לכל** היותר יכולם להתקיים במערכת ברגע כלשהו?

- ללא התחשבות בתהיליך-hellshell עצמו או תהליכי אחרים שאינם נתונים בשאלת.

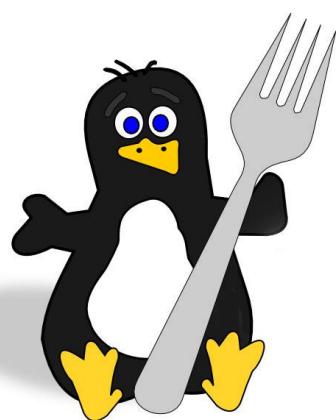
א. 1      ב. 2      ג. N      ד. N+1      ה. 2N

- ומה התשובה אם מרכיבים את תוכנית B בלולאה?

The answer here is the same for both processes: A total of  $2N$  processes are possible – **N fathers, and N sons**,

The reason is that the shell continues to the next iteration immediately, without waiting for the previous iteration to finish.

## הפוך



i don't give a fork

## ניהול תהליכי בגרעין לינוקס

כדי לקשר את הפרק זהה לפרק הקודם, כדאי לשאול את הסטודנטים באילו מבני נתונים של הגרעין משתמשות קריאות המערכת השונות.  
למשל, (`getpid()` ניגשת רק למתאר התהליך הנוכחי וקוראת את השדה `pid`. היא לא ניגשת לרשימת התהליכים הכלובאלית או לטבלת המיפוי ההפוך.

למעשה, אני חושב שענייף להציג את הפרק כולו באופן שונה ולדבר על המימוש של קריאות המערכת ... `exit`, `fork`, `wait`, במקומות לדבר על מבני הנתונים.  
למשל: במקום לדבר על תורי המתנה, להציג את המימוש של `wait` ומשם להגיע באופן טבעי לתורי המתנה.

## מבנה הנטוניים לניהול תהליכיים

- גראין לינקס מימוש את קרייאות המערכת שראינו (...).  
באמצעות מבני הנטוניים הבאים:
  - מתאר התהלייך** (Process Descriptor).
  - במערכות הפעלה אחרות נקרא PCB = Process Control Block.
  - שומר בתוכו גם קשרי משפחה.
- רשימת התהלייכים** (Process list).
- טבלת ערבול PID → PCB**.
- טור ריצה ("הטוווח הקצר") – Run Queue.
- טור המתנה ("הטוווח הבינוני / אחר") - Waiting Queue.

## מתאר התהיליך

- לכל תהיליך בLINUKS קיימ בגרעין מתאר תהיליך ( `process` ), שהוא אובייקט מטיפוס `task_struct` המכיל את:
  - מזהה התהיליך (PID).
  - מצב הריצה של התהיליך.
  - עדיפות התהיליך.
  - מצביעים למתאר תהיליך האב ו"קרובי משפחה" נוספים.
  - מצביע ל לבטל אזרוי הזיכרון של התהיליך.
  - מצביע ל לבטל הקבצים הפתוחים של התהיליך.
  - מצביעים למתאר תהיליכים נוספים (רשימה מקוشرת).
  - מסוף אליו התהיליך מתקשר.
  - ועוד...

The struct `task_struct` is defined in `include/linux/sched.h` .

## מימוש קראת המערכת (pid)

- מוגדרת בקובץ . kernel/timer.c

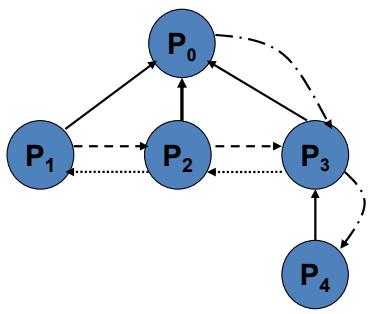
```
long sys_getpid(void) {
 return current->pid;
}
```

current הוא מצביע  
למצביע התהיליך הנוכחי.

### Advanced Notes:

- The actual implementation returns **current->tgid**, but we still haven't learned about threads at this point in the course.
- The function returns **long** for 64 compatibility. It was once an int :  
<https://stackoverflow.com/questions/20940212/why-is-linux-syscall-return-type-long>

## ניהול קשרי משפחה בגרעין



—————→ (real\_parent)  
 -----→ siblings  
 -----→ children

- "קשרי המשפחה" בין תהליכיים נשמרים באמצעות מצביעים הנשמרים ב-PCB.

1. `real_parent`: מצביע לאב המקורי.

2. `parent`: מצביע לאב בפועל.

- האב בפועל שונה מהאב המקורי כאשר התהילך נמצא בריצה מבוקרת, למשל תחת debugger.

- כרגע תחיליך יכול לאתור את אבי, למשל עבור קריית המערכת (`getppid()`).

שאלת של סטודנט: האם כדי לתאר בן יתומםoppo מצביע ל-NULL ו-pptr מצביע לתהיליך init שירש בנימין יתום?

תשובה: ע"פ הספר UTLK3 התשובה שלילית. opptr, pptr יצביעו שניהם על init.  
`real_parent [opptra]` - Points to the process descriptor of the process that created P or to the descriptor of process 1

(init) if the parent process no longer exists. (Therefore, when a user starts a background process and exits the shell, the background process becomes the child of init.)

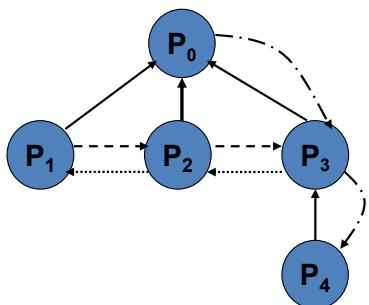
### Real parent vs. parent

The field parent in task\_struct usually matches the process descriptor pointed by `real_parent`.

`real_parent`: Points to the process descriptor of the process that **created** P or to the descriptor of process 1 (**init**) if the parent process no longer exists.

`parent`: Points to the **current** parent of P (this is the process that must **be signaled** when the child process terminates, i.e. SIGCHLD). It may occasionally differ from `real_parent` in some cases, such as when another process issues a `ptrace()` system call requesting that it be allowed to monitor P.

## ניהול קשרי משפחה בגרעין



- (real\_parent)
- siblings
- children

3. `children`: מצביע לרשימה מקושרת של בניים.

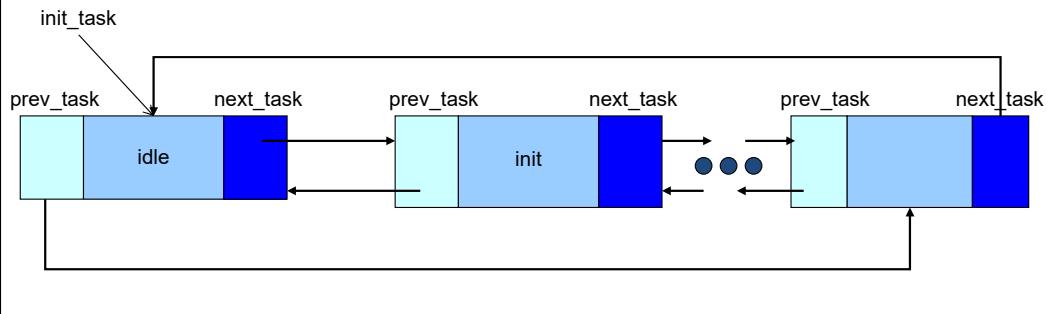
4. `siblings`: מצביע לרשימה מקושרת של אחים, (תהילכים הנוצרים ע"י אותו תהילך אב).

- כך תהילך יכול לאתר את בניו לפי סדר יצרתם, למשל עבור קראית המערכת `(wait)`.

שימוש לב: התיעוד של `wait()` לא מבטיח ערך חוזה לפי סדר הייצרה, למרות שכך לינוקס עושה בפועל.

## רשימת התהליכים

- מתאר כל התהליכים מחוברים ברשימה **מקושרת כפולה מעגלית** (linked list) באמצעות השדות `next_task` ו-`prev_task`.
- ראש הרשימה הוא המתאר של התהליך `idle` (מצבע ע"ז).
- שאלה:** למה הגיוני לעשות רשימה זו מקושרת **כפולה**?



**Answer:** it is useful for removing and adding mid list – we can supply a pointer to one of the list elements and use it to delete the cell and connect the next process to the one before the removed cell.

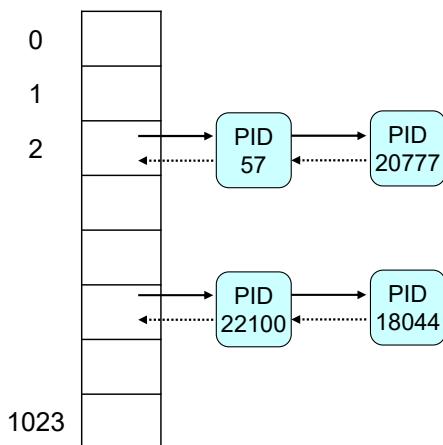
**Question for the students:** Why do we need this, if we have the hash table?

**Answer:** For operations that need to iterate over all of the available processes – for example, PID recycling.

## פעולת על רשימת התהליים

- איזו קריית מערכת מוסיפה איבר (PCB) לרשימת התהליים?
  - (fork, כי היא יוצרת תהלים חדש.
  - שימוש לבן execv אינה יוצרת תהלים חדש ולכן אינה מוסיפה איבר לרשימה.
- 
- איזו קריית מערכת מוחקת איבר (PCB) מרשימת התהליים?
  - (wait, כי היא מוחקת תהלים שהסתיים.
  - שימוש לבן exit אמן מסיימת תהלים אבל אינה מוחקת אותו לגמרי (התהלים עובר למצב זומבי).

## טבלת ערבול PID → PCB



- חלק מקריאות המערכת, למשל `waitpid()`, מתייחסות לתהליכיים ע"פ ה-PID שלהם.

- **בעיה:** חיפוש תחילה לפי PID ברשימה המקורית של כל התהליכים הוא בסיבוכיות ( $O(n)$ ) כאשר  $n$  הוא מספר התהליכים במערכת.

- **פתרון:** הגרעין שומר טבלת ערבול (hash table) המאפשר לאתר תחילה לפי PID שלו בסיבוכיות  $O(1)$  בממוצע.

- גודל הטבלה הוא `Z_SZ_HASH_PID` (בד"כ 1024) כניסות.
- בדרך כלל מספר התהליכים במערכת קטן בהרבה מ-2K וכאן אין צורך להחזיק כניסות עבור כל ה-PID האפשרי.
- התנטשויות בטבלה היגיוב נפתרות בשיטת separate chaining, כלומר תהליכיים שונים עברו רם פונקציית `hash` מחזירה ערך זהה נמצאים ברשימה מקוישת השומרת את מתאריו התחילה.
- שרשרת ההתנטשויות בכל תא בטבלה היא רשימה מקוישת דו-כיוונית.

## תורי תהליכיים

תור ריצה לכל מעבד  
תור המתנה לכל אירוע המתנה

תהליכיים בתורי הריצה נקראים לפעמים "תהליכיים בטווח הקצר" ואילו תהליכיים בתורי המתנה נקראים "תהליכיים בטווח הבינוני/ארוך".  
זמן התהליכיים לטוח הקצר בוחר תהליכיים מטור הריצה למעבד. זמן התהליכיים לטוח הבינוני/ארוך מעביר תהליכיים מטור המתנה לתורי הריצה.

עוד מידע על זמן לטוח הקצר/בינוני/ארוך:  
<http://techdifferences.com/difference-between-long-term-and-short-term-scheduler.html>  
<https://www.cim.mcgill.ca/~franco/OpSys-304-427/lecture-notes/node38.html>

מצב התהילה

- מצב התהיליך נשמר בשדה **state** במתאר התהיליך.
  - משתנה בגודל 32 ביט המתפרק כמערך ביטים: בכל רגע נתון, **בדיקה אחד מהביטים** ב-state **דולק** בהתאם למצב התהיליך באותו זמן.

0000 0000 0000 0000 0000 0000 0000 0001

- המצביעים האפשריים לתהיליך בLINQoS הם:
    1. **TASK\_RUNNING** – התהיליך רץ או מוכן לריצה.
    2. נאמר כי התהיליך יזום לריצה "בטווח הקצר".
  - עדין לא ביקש מידע על סיום התהיליך הסתיימה, אך התהיליך האב של התהיליך – **TASK\_ZOMBIE** – ריצת התהיליך הסתיימה, אך באמצעות קריאה כדוגמת (`wait()`)
  - מタוך התהיליך הוא הדבר היחיד שנוטר ממנו.

את ערך השדה state ניתן לשנות בהצבה ישירה או על-ידי המאקרו `set_task_state` או `set_current_state` (קובץ גרעין `include/linux/sched.h`).

## מצב התהיליך

### .3. המתנה "RDDOA" – **TASK\_INTERRUPTIBLE**

- התהיליך **ממתן** לאירוע כלשהו אך ניתן להפסיק את המתנה התהיליך ולהחזירו למצב הנפוץ.
- זהו מצב המתנה הנפוץ. **מתי נמצא תהיליכים במצב זה?**

• דוגמה 1: תהיליך אב הממתן לסיום הבן (קריאה מערכת `wait`).

• דוגמה 2: דפדפן (web browser) מוחכה לקבלת נתונים מהרשת (דף web) אבל אפשר לקטוע את המתנה עלי-ידי סגירת חלון הישום, שגורמת לשילוח אות לסיום התהיליך.

### .4. המתנה "עמוקה" – **TASK\_UNINTERRUPTIBLE**

- התהיליך **ממתן** לאירוע כלשהו אך לא ניתן להפסיק את המתנה התהיליך באמצעות שליחת סיגナル לתהיליך.
- מצב המתנה נדר.

• דוגמאות: בשולי השקופית – דורשות חומר מתקדם בקורס.

**.5. TASK\_STOPPED** – ריצת התהיליך נעצרה בצורה מבוקרת על-ידי תהיליך אחר (בדרכ-כלל debugger או tracer).

עוד לא למדנו על שליחת סיגנלים – נגיע לכך בתרגול 7.

TASK\_INTERRUPTIBLE, the interruptible sleep. If a task is marked with this flag, it is sleeping, but can be woken by signals. This means the code which marked the task as sleeping is expecting a possible signal, and after it wakes up will check for it and return from the system call. After the signal is handled, the system call can potentially be automatically restarted. TASK\_UNINTERRUPTIBLE, the uninterruptible sleep. If a task is marked with this flag, it is not expecting to be woken up by anything other than whatever it is waiting for, either because it cannot easily be restarted, or because programs are expecting the system call to be atomic. This can also be used for sleeps known to be very short.

Uninterruptable processes are USUALLY waiting for I/O following a page fault.

Consider this:

The thread tries to access a page which is not in core (either an executable which is demand-loaded, a page of anonymous memory which has been swapped out, or a mmap()'d file which is demand loaded, which are much the same thing)

The kernel is now (trying to) load it in

The process can't continue until the page is available.

The process/task cannot be interrupted in this state, because it can't handle any signals; if it did, another page fault would happen and it would be back where it was.

קריאה למקדים:

<https://stackoverflow.com/questions/223644/what-is-an-uninterruptable-process>

## טור ריצה (runqueue)

- התהליכים המוכנים ל裏צה (מצב TASK\_RUNNING) נשמרים במבנה נתונים הקרי **runqueue** (טור ריצה).

- לכל מעבד יש טור ריצה (מבנה `runqueues`) משלו:

```
struct rq runqueues[NR_CPUS];
```

- בכל רגע נתון, תהליך יכול להימצא בתור ריצה אחד לכל היוטר.

מודיע?



- מבנה הנתונים של טור ריצה מורכב מ:
  - מערך של תורים לתהליכי זמן-אמת.
  - עכ דום-שחור לתהליכי רגילים.
- פרטים נוספים בתרגול על שימוש בתהליכיים.

תשובה: מערכת הפעלה בוחרת את התהליך הבא ל裏צה על המעבד מתוך טור הריצה של אותו מעבד.

אם תהליך מסוים היה נמצא בשני טורי ריצה של שני מעבדים שונים באותו רגע, הוא יהיה עלול להיקרא ל裏צה על שני המעבדים בו-זמןית, וזה כמובן אסור.

(שים לב: ניתן להריץ את אותה תוכנית בו-זמןית על שני מעבדים שונים בשני תהליכיים שונים. אבל תהליך אחד יכול לירוץ רק על מעבד אחד בכל רגע נתון).

The accurate definition of runqueues can be found in kernel/sched/sched.h :

```
DECLARE_PER_CPU_SHARED_ALIGNED(struct rq, runqueues);
```

## פעולות על תור ריצה

- הfonקציות `activate_task()`, `deactivate_task` מכניות ומציאות תהליכי מטור ריצה, בהתאם.
- שימוש אפשרי לדוגמה: הפונקציה `wake_up_process()` הופכת תהליך ממתיין (למשל במצב `TASK_INTERRUPTIBLE`) לモון לריצה (מצב `TASK_RUNNING`):
  - .1 מוצאת את תור הריצה של המעבד הנוכחי.
  - .2 מוסיף את התהליך לתור הריצה זהה באמצעות `.activate_task()`.
  - .3 מסמנת צורך בהחלפת הקשר אם התהליך החדש בעדיפות גבוהה יותר מההתהליך שרצ כרגע על המעבד.

## טור המתנה (wait queue)

- תחיל'ר שממתין לארוע כלשהו (מצבים TASK\_INTERRUPTIBLE או TASK\_UNINTERRUPTIBLE) נמצא בטור המתנה (ואינו נמצא באף תור ריצה).
- לכל סוג אירוע יש טור המתנה נפרד, לדוגמה:
  - טור המתנה לכל סוג של פסיקת חומרה, למשל דיסק או שעון.
  - טור המתנה לכל משאב מערכת שיתפונה לשימוש. לדוגמה: ערוץ תקשורת שיתפונה כדי לשЛОח דרכו נתונים.
  - טור המתנה לכל תחיל'ר עבר סיום אחד הבנים שלו.
- תחיל'ר יכול לעبور לטור המתנה רק באמצעות קריית מערכת חוסמת (למשל ... read, wait, ...) אשר מוגדרת (yield) על המעבד.

**טור המתנה שבו תחיל'ר נמצא ומחכה לסיום הבנים שלו נשמר בשדה:**

**wait\_queue\_head\_t wait\_chldexit;**

של task\_struct של אותו תחיל'ר (כלומר, תחיל'ר האב).  
שימוש לב שזה "טור" מאד מנומן, מפני שהוא יכול לכל היותר תחיל'ר אחד בכל רגע נתון.

## תרגול 3

---

סיג널ים (Signals)  
קלט/פלט של תהליכיים  
תקשורות בין תהליכיים

## TL;DR

- תהליכיים בלינוקס יכולים לתקשר ביניהם במגוון אמצעים, למשל:

| pipes        | מערכת הקבצים                   |
|--------------|--------------------------------|
| >> ls   more | >> ls > temp<br>>> more < temp |

- לינוקס מציגו ממשק **אחד** לכל אמצעי התקשרות בעזרת קבצים.
  - מימוש של גישת "Everything is a file" שהומצאה ביוניקס.
  - אמצעי תקשורת - קבצים רגילים, התקני קלט/פלט (מסך, מקלדת, עכבר, ... ) וערוצי תקשורת ייעודיים כמו pipes, sockets, ...).
- בנוסף, לינוקס מאפשרת תקשורת מינימליסטית בין תהליכיים באמצעות סיג널ים (**signals**) – אותן מספריים שלמים בין 1—31.

חסרונות התקשרות באמצעות קבצים: הגישה לדיסק היא איטית, ובנוסף יש קובץ זמן שציריך למחוק.

## מנגנוני IPC בLinux

- בדומה למערכות הפעלה מודרניות אחרות, Linux מציע מגוון מנגנונים לתקשורת בין תהליכים:
  - (באנגלית: IPC = Inter-Process Communication).
- signals:** הודעות אסינכרוניות הנשלחות בין תהליכים באמצעות מכונה (וגם מערכת הפעלה לתהליכים) על-מנת להודיע לתהליך המתקבל על אירוע מסוים.<sup>1</sup>
- pipes, FIFOs:** ערכז תקשורת בין תהליכים באמצעות מכונה בסגנון יצרן-צרכן.<sup>2</sup>
- sockets:** המנגנון סטנדרטי לייצור ערז תקשורת דו-כיווני בין תהליכים היוכלים להימצא גם במכונות שונות. משמש לתקשורת בראשת האינטרנט.<sup>3</sup>

## מנגנוני IPC של V System

- לינוקס תומכת גם במנגנוני תקשורת וסנסרין של V System.
- אוסף מנגנוני תקשורת שהופיעו לראשונה בגרסאות UNIX של חברת AT&T ואומצאו במהרה על-ידי מרבית מערכות ההפעלה מבוססות ה-UNIX.
- המנגנוןים של V System הם:
  - סمفוריים.
  - תורי הודעות (message queues) – מנגנון המאפשר להגדיר "תיבות דואר" וירטואליות הנגישות לכל התהליכים אותה מכונה. תהליכי יכולם לתקשר באמצעות הכנסה והוצאה של הודעות מאותה תיבת דואר.
  - זיכרון משותף – יצרת איזור זיכרון מיוחד המשותף למרחבי הזיכרון של מספר תהליכים. זו צורת התקשרות הייעילה ביותר והמקובלת ביותר עבור ישומים הדורשים העברת מידע רב.
- לא נסקרו את המנגנוןים האלה בתרגול.

## סיגנלים (SIGNALS)

---

## סיג널ים (signals)

- מנגנון לשילוחה וודעות לתהליכיים.
- גם (1) בין תהליכיים, וגם (2) בין מערכת הפעלה לתהליכיים.
- המנגנון ממומש בתוכנה בלבד, ללא תמיכת חומרה.
- סיג널ים נשלחים באופן **אסינכרוני** ויכולים להגיע בכל נקודה בזמן.
- אירוע אסינכרוני == אירוע חיצוני לקוד המשתמש אשר קוטע את ריצת התוכנית וגורם למעבד להתחילה בצד את שגרת הטיפול באירוע.
- תחילר לא צריך לקרוא או להמתין לסיג널ים, הסיג널ים פשוט "מגעים" לתחילר.
- **שימוש לב: סיג널ים ≠ פסיקות** (טעות נפוצה של סטודנטים).
- המקור לטעות הוא (כנראה) שגם פסיקות הן אירועי אסינכרוני.  
אבל בኒיגוד לפסיקות, סיג널ים מטופלים במצב משתמש (user mode).

```

#define SIGHUP 1
#define SIGINT 2
#define SIGQUIT 3
#define SIGILL 4
#define SIGTRAP 5
#define SIGABRT 6
#define SIGBUS 7
#define SIGFPE 8
#define SIGKILL 9
#define SIGUSR1 10
#define SIGSEGV 11
#define SIGUSR2 12
#define SIGPIPE 13
#define SIGALRM 14
#define SIGTERM 15
...

```

**בלינוקס יש 31 סיגנלים,  
לכל אחד שם ומספר  
שלם בין 1—31**

המשתמש לחץ על CTRL+C ב-shell –  
תהליך-h-shell ישלח SIGINT לתהליך  
שרץ בחזית.

תהליך ביצע פקודה לא חוקית – מערכת  
הפעלה תשלח לתהליך SIGKILL.

תהליך ניגש לכתובת לא חוקית בזיכרון –  
מערכת הפעלה תשלח לתהליך SIGSEGV.

רשימת הסיגנלים המלאה מוגדרת בקובץ:  
 /usr/include/asm-generic/signal.h  
 לא קיים סיגナル 0, איך שליחה שלו משמשת לבדיקת שגיאות על תהליכי שליחת הסיגנלים.

## קריאה המערכת **kill**

```
#include <sys/types.h>
#include <signal.h>
int kill(pid_t pid, int sig);
```

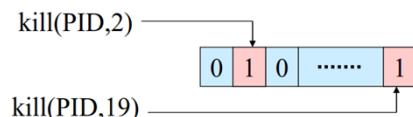
- פעולה: שולחת את הסיגナル שמספרו **sig** לתהיליך המזוהה ע"י **pid**.  
אם **0==sig**, אז הפעולה רק בודקת שהטהיליך **pid** קיים מבל' לשולוח **signal**  
(שימושי לבדיקה תקיפות **pid**).
- ערך מוחזר: 0 בהצלחה,  
- בכישלון (למשל אם אין תהיליך בעל מזהה **pid**).

שימוש לב: השם של קריית המערכת (**kill** מבלב), מפני שהוא משמשת לשילוח כל סוג  
הסיגנלים ולא רק **SIGKILL**.  
פקודת **kill** בلينוקס קוראת למעשה לקריית המערכת (**kill**).

## העברה סיג널ים בשני שלבים

**1. רישום –** מערכת הפעלה רושמת ב-PCB של תהליך היעד שיש לו סיגナル ממתיין (**pending signal**).

- הרישום מתבצע במערך **ビナרי** בין 31 ביטים, ולכן יכול להיות לכל היותר סיגナル ממתיין אחד מכל מספר.



**2. טיפול –** בכל פעם שהתהליך חוזר במצב גרעין **למצב משתמש**, המערכת הפעלה בודקת אם יש סיג널ים ממתיינים ומטפלת בהם.

- בסוף הטיפול בסיגナル, המערכת הפעלה תאפס את הביט המתאים במערך.
- במידה ויש מספר סיגナルים ממתיינים, סדר הטיפול מתחילה מחדש.

המונחים באנגלית (כפי שמופיעים ב-page man) מבלבלים לטעמי: רישום == generation, טיפול == delivery.

רישום של סיגナル גורם **לסיום המתנה** של תהליך במצב TASK\_INTERRUPTIBLE ויציאה מה-.waitqueue.

במילים אחרות, תהליך שהמתין במצב TASK\_INTERRUPTIBLE בעקבות קראית מערכת חוסמת, וקיים סיגナル תור כד', יחזיר למצב משתמש עם תוצאה כישלון EINTR- עקב הפרעה. הסיגナル קוטע את ביצוע קראית המערכת.



## טיפול בסיג널ים

- תהיליך יכול לטפל בסיגナル במספר אופנים, לדוגמה:

- .1 – סיום התהיליך בתגובה לסיגナル.
- .2 – התעלמות מהסיגナル והמשך הביצוע הרגיל.
- .3 – עצירת התהיליך במצב TASK\_STOPPED (בד"כ בשליתת debugger).
- .4 – המשך ביצוע התהיליך שהיה במצב TASK\_STOPPED (בד"כ בשליתת debugger).
- .5 – **"תפיסת הסיגナル"** (catching signals) – הפעלת שגרת משתמש מיוחדת (signal handler) בתגובה לסיגナル.

## קריאה המערכת (`signal()`)

```
typedef void (*sighandler_t)(int);
sighandler_t signal(int signum,
 sighandler_t handler);
```

- פעולה: משנה את אופן הטיפול בסיגナル שמספרו `signum`.

- פרמטרים:

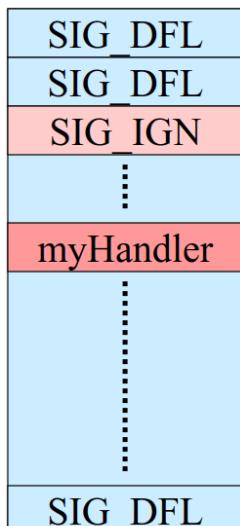
- `signum` – מספר בתחום 1—31 פרט ל-`SIGKILL` ו-`SIGSTOP` (9).
- `handler` – מצביע לפונקציית משתמש או `SIG_IGN` או `SIG_DFL`.

- ערך מוחזר:

- בהצלחה, ערכו הקודם של ה-`signal` (פונקציה קודמת / `SIG_DFL` / `SIG_IGN`).
- בכשלון, `SIG_ERR`.

**Some functions are not safe to call from within a signal handler**, such as `printf`, `malloc`, etc. A useful technique to overcome this is to use a signal handler to set a flag and then check that flag from the main program and print a message if required. Further reading: <http://www.ibm.com/developerworks/linux/library/l-reent/index.html>

## המבנה signal\_struct



- ב-PCB שמור מבנה בן 31 תאים ובו שמורות פעולות הטיפול בכל סיגנל.

- אופציות לטיפול:

- .1 SIG\_DFL – בצע את טיפול ברירת המחדל בסיגנל זה.
- .2 SIG\_IGN – התעלם מהסיגナル.
- .3 קישור ל-signal handler שהוגדר ע"י המשתמש.

לדוגמה: ברירת המחדל לאופן הטיפול ב-SIGFPE היא dump. באמצעות הקראיה (`(signal)`) יכול המתכנת להחליף את אופן הטיפול בהפעלת signal handler שתדפיס הודעה שגיאת ותמשיך את הביצוע הרגיל של התכנית (שים לב שהמסכת התכנית במצב זה מוגדרת כת-undefined behavior ולכן אין סיבה הגיונית לעשות את זה).



## שגרות טיפול בסיג널ים

- תהילך יכול להתקין שגרת טיפול בסיגナル (signal handler) שתיקרא בעת קבלת סיגナル מסוים.
- השגרה מבוצעת ב-**mode user**, בהקשר של התהילך שקיבל את הסיגナル.
- ניתן להתקין שגרת טיפול חדשה לכל סיגナル פרט ל-SIGSTOP (9) ו-SIGKILL (17).
- איך מגנים על תהילך שהריץ קוד וקיבל סיגナル?
- הקשר הבינלאומי של התהילך נשמר לפני התחלת ביצוע השגרה ומשוחזר לאחר סיוםה, אך הפעלת השגרה לא גורמת להחלפת הקשר התהילך.
- במהלך ביצוע השגרה נחסם זמנית (masking) טיפול בסיג널ים מהסוג שגורם לביצוע השגרה, על מנת למנוע בעיות של reentrancy.

## דוגמה

```
>> gcc signal.c
>> a.out &
[1] 3189
Waiting...
>> kill 3189
Hi
Bye
[1]+ Done a.out
>>
```

שליחת סיגנל  
**SIGTERM**  
 מסוג kill  
 באמצעות kill  
 (bash utility)

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

void catcher1(int signum) {
 printf("Hi\n");
 kill(getpid(), 22);
}

void catch22(int signum) {
 printf("Bye\n");
 exit(0);
}

main() {
 signal(SIGTERM, catcher1);
 signal(22, catch22);
 printf("Waiting...\n");
 while(1);
}
```

### Important Questions to ask:

First question to ask: When will Process 3189 see the SIGTERM waiting signal?

**Answer:** At the end of the next interrupt

Second question to ask: When will Process 3189 see signal 22 (SIGPOLL)?

**Answer:** Right after the finishing the kill system call.

## שליחת סיג널ים בין תהליכי

- שימוש נפוץ בסיגナル הינו בלימה של ביצוע תהליך ע"י המשתמש.
  - למשל בתחום ה-**shell**:
    - לחיצה על CTRL+C גורמת לשילוח SIGINT לתהליך. טיפול ברירת המחדל בסיגナル זה הינו סיום התהליך.
    - לחיצה על CTRL+Z גורמת לשילוח SIGTSTP לתהליך. טיפול ברירת המחדל בסיגナル הינו השהייה של ריצת התהליך.

מי התהליך שאחראי על  
שליחת הסיג널ים?

תשובה: ה-**shell**.

(from: <https://unix.stackexchange.com/questions/362559/list-of-terminal-generated-signals-eg-ctrl-c-sigint>)

## שליחת סיגלים ע"י מערכת הפעלה



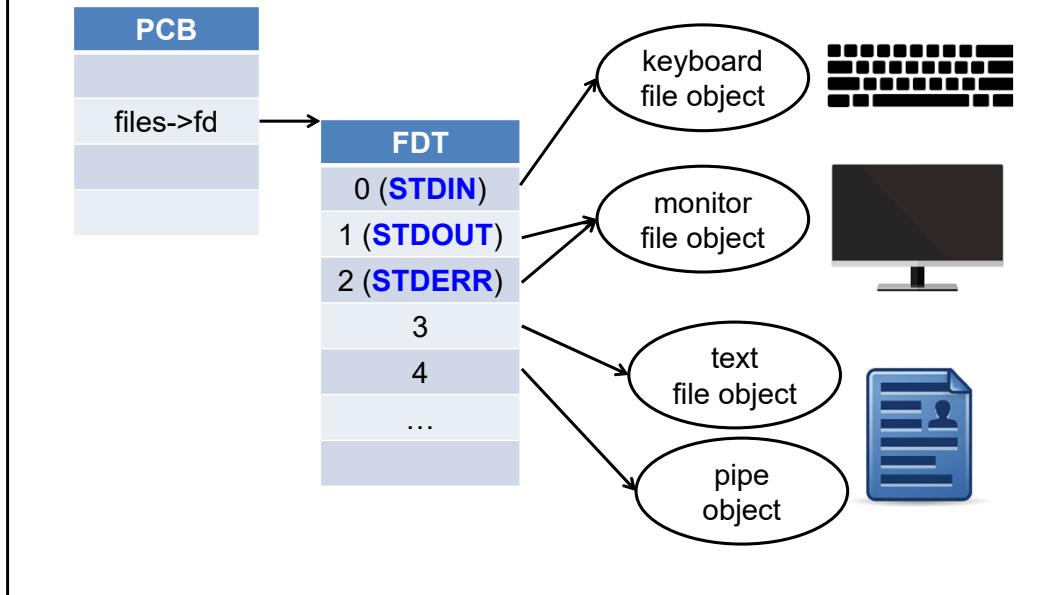
## סיגנלים – סיכום

- סיגנלים נשלחים בשני שלבים: (1) רישום, (2) טיפול.
  - מערכת הפעלה  $\leftarrow$  תהלייר:
    - תהלייר B יצר חריגה הדורשת את התערבות ממערכת הפעלה.
    - מערכת הפעלה מדיעה לתהלייר B על האירוע ע"י רישום סיגנל ב-PCB שלו.
    - במעבר מצב גרעין לקוד משתמש של תהלייר B, יטופל הסיגナル.
  - תהלייר  $\leftarrow$  תהלייר:
    - תהלייר A פונה למערכת הפעלה שתרשום סיגナル לתהלייר B.
    - מערכת הפעלה רושמת סיגナル ב-PCB של תהלייר B.
    - במעבר מצב גרעין לקוד משתמש של תהלייר B, יטופל הסיגナル.

## קלט/פלט של תהליכיים

---

## Everything is a file!



## FD (file descriptors)

- כל פעולות קלט/פלט של תהיליך בLINUKS מבוצעות דרך "קבצים":
  - קבצים "רגילים" לאחסון מידע (usr/file.txt) נמצאים **בדיסק**.
  - התקני חומרה גם כן מיוצגים כקבצים, אבל נמצאים **בזיכרון**.
  - למשל, העכברים המוחברים למחשב מיוצגים כ- /dev/input/mouse0 .
  - גם ערכוי תקשורת כמו סופוק מיווצגים ע"י קבצים שנמצאים בזיכרון.
- הקשר בין תהיליך לבין קובץ שהוא ניתן נשמר, בرمת המשמש, ע"י מספר שלם שנקרא **file descriptor (FD)**.
  - לדוגמה: קריאת המערכת ()open FD מחזירה FD.
  - המשמש מעביר את ה-FD לקריאות מערכת כמו ()write, ()read כדי לקרוא ולכתוב לקובץ.

המציאות, כמובן, טיפה מורכבת יותר:

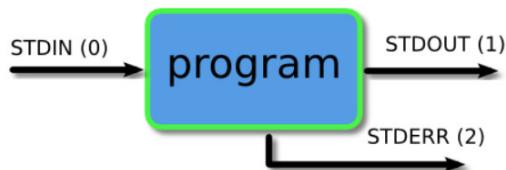
<https://unix.stackexchange.com/questions/204977/why-isnt-monitor-listed-under-dev-in-linux>  
<https://unix.stackexchange.com/questions/25601/how-do-mouse-events-work-in-linux>

## FDT (file descriptor table)

- ברמת הגרעין, FD הוא אינדקס לכינסה בטבלה הנקראת **(FDT) file descriptor table**.
- לכל תהליך יש FDT מסויל, המוצבעת ע"י השדה `fd_files` ב-PCB.
- כל כניסה ב-FDT מחייבת על **אובייקט ניהול קובץ פתוח (file object)**.
- אובייקט הניהול נגיש וمتוחזק ע"י גרעין מערכת הפעלה בלבד.
- object 파일 מכיל מספר שדות.
- למשל את "מחוון הקובץ" (**seek pointer**) המצביע למקום הנוכחי בקובץ (כלומר, מהין לקרוא/לכתוב את הנתונים הבאים).
- כל הקבצים הפתוחים של כל התהליכים במערכת נשמרים גם הם בטבלה **globaalit mnogalat** ע"י הגרעין – ה-GDT – או GFDT.

## ערוצי הקלט/פלט הסטנדרטיים

- ערכי ה-FD הבאים מוקשרים לתוכנים הבאים כברירת מחדל:
  - 0 – ערוץ הקלט הסטנדרטי (**STDIN**), בדרך כלל מוקשור למקלדת.
  - פעולות הקלט המוכרות, כדוגמת (`scanf` ודומותיה, קוראות למשה מהתקן הקלט הסטנדרטי).
- 1 – ערוץ הפלט הסטנדרטי (**STDOUT**), בדרך כלל מוקשור למסך.
  - פעולות הפלט המוכרות, כדוגמת (`printf` ודומותיה, כתובות למשה להתקן הפלט הסטנדרטי).
- 2 – ערוץ השגיאות הסטנדרטי (**STDERR**), בדרך כלל גם הוא מוקשור למסך.



## דוגמת קיד

```
>> cat main.c
#include <stdio.h>

int main() {
 printf("Hello World!\n");
 FILE* f = fopen("file.txt", "w");
 fprintf(f, "Hello World!\n");
 return 0;
}

>> gcc main.c
>> strace ./a.out
...
write(1, "Hello World!\n", 13) = 13
open("file.txt", O_WRONLY|O_CREAT|O_TRUNC, 0666) = 3
write(3, "Hello World!\n", 13) = 13
...
```

מאחורי פונקציות libc אלו  
מסתורות קריאות המערכת  
open(), read(), write()

strace הוא כלי המאפשר לעקוב אחר  
קריאות מערכת במהלך התוכנית

1 = STDOUT

13 = Number of chars written

3 = The FD of file.txt

## פתיחת קובץ לגישה

```
#include <sys/types.h>
#include <sys/stat.h>
```

איך יתכונו שתי חתימות שונות של פונקציות עם אותו שם ב-C?

```
#include <fcntl.h>
int open(const char *path, int flags);
int open(const char *path, int flags,
 mode_t mode);
```

- פעולה: פותחת את הקובץ המבוקש (לפי path) לגישה לפי התוכנות המוגדרות ב-flags ולפי הרשות המוגדרות ב-mode.

### ערך מוחזר:

- במקרה של הצלחה – ה-FD הקשור לקובץ שנפתח.

- **האינדקס המוקצה בטבלה** הוא האינדקס הפנוי הנמור ביותר ב-FDT.

- במקרה של כישלון – (-1).

העשרה:

כל הקצאה של כניסה ב-FDT עוברת דרך אותה פונקציה בגרעין האחראית על מציאת הכניסה הפנوية הראשונה בטבלה. הפונקציה הזאת משתמשת בפקודות מכונה מיוחדות כדי להאיץ את החיפוש (בדומה לחיפוש התהיליך הבא ליריצה בפונקציה `(schedule()`).

## פתיחת קובץ לגישה (2)

- ההגדרה האמיתית של `()open()` נראה ככך:

```
int open(const char *path, int flags, ...);
```

- `open()` מקבלת מספר משתנה של פרמטרים בדומה ל`-printf()`.

- פונקציות מסווג זה (variadic functions) מוגדרות עם הסימן "... ברישיות הארגומנטים.

- הารגומנט השלישי (`mode`) יקרא ע"י `open()` רק אם הארגומנט השני

- (`flags`) מאפשר **יצירת קובץ חדש**.

- ולכן אם מגדירים `יצירת קובץ חדש, חייבים להעביר את הארגומנט השלישי.`

From `open(2)`:

`mode` specifies the permissions to use in case a new file is created. This argument must be supplied when `O_CREAT` is specified in `flags`; if `O_CREAT` is not specified, then `mode` is ignored. The effective permissions are modified by the process's `umask` in the usual way: The permissions of the created file are  $(mode \& \sim umask)$ . Note that this mode only applies to future accesses of the newly created file; the `open()` call that creates a read-only file may well return a read/write file descriptor.

## פתיחת קובץ לגישה (3)

### פרמטרים:

- path – מסלול לקובץ (או התקן) לפתיחתו. לדוגמה:
  - לציין הקובץ file1" בספרית העובדה הנוכחית.
  - לציין קובץ בכתובת אבסולוטית.
- – תכונות לאפיון פתיחת הקובץ. חייב להכיל אחת מהאפשרויות הבאות:
  - O\_RDONLY – הקובץ נפתח לקריאה בלבד.
  - O\_WRONLY – הקובץ נפתח לכתיבה בלבד.
  - O\_RDWR – הקובץ נפתח לקריאה ולכתיבה.
- ניתן להוסיף תכונות אופציונליות באמצעות OR (|) עם הדגל המתאים, למשל:
  - O\_CREAT – צור את הקובץ אם אינו קיים.
  - O\_APPEND – שרשר מידע בסוף קובץ קיים.
- mode – פרמטר אופציוני המגדיר את הרשאות הקובץ, במקרה שפתחת הקובץ גורמת לצירף קובץ חדש (למשל, כאשר flags מכילים O\_CREAT).

### .(mode & ~umask)

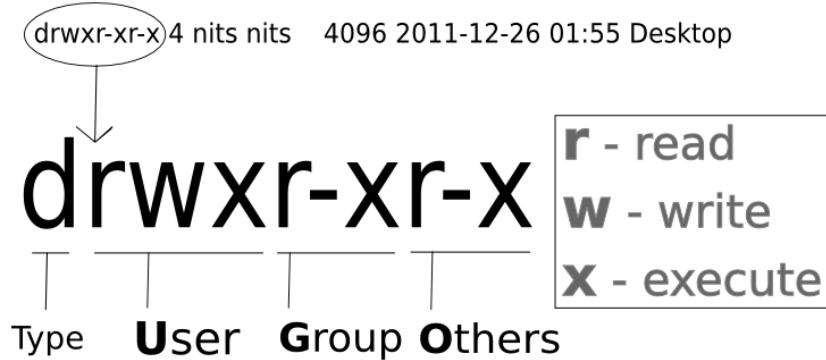
The umask acts as a set of permissions that applications cannot set on files. It's a file mode creation mask for **processes** and cannot be set for directories itself. Most applications would not create files with execute permissions set, so they would have a default of 666, which is then modified by the umask.

As you have set the umask to remove the read/write bits for the owner and the read bits for others, a default such as 777 in applications would result in the file permissions being 133. This would mean that you (and others) could execute the file, and others would be able to write to it.

If you want to make files not be read/write/execute by anyone but the owner, you should use a umask like 077 to turn off those permissions for the group & others. In contrast, a umask of 000 will make newly created directories readable, writable and descendible for everyone (the permissions will be 777). Such a umask is highly insecure and you should never set the umask to 000.

The default umask on Ubuntu is 022 which means that newly created files are readable by everyone, but only writable by the owner:

## הרשאות קבצים בLinux



*File permissions in Linux*

Taken from: <https://nitstorm.github.io/blog/understanding-linux-file-permissions/>

## הרשאות קבצים בلينוקס – דוגמה

```
[idanyani@csm ~/Downloads]$ ls -l -h
total 210M
drwxr-xr-x 6 idanyani assist 8.0K Dec 13 2016 jre1.8.0_121
drwxr-xr-x 6 idanyani assist 8.0K Dec 20 2017 jre1.8.0_161
drwxr-xr-x 6 idanyani assist 8.0K Apr 10 2015 jre1.8.0_45
-rw-r--r-- 1 idanyani assist 71M Feb 8 2017 jre-8u121-linux-x64.tar.gz
-rw-r--r-- 1 idanyani assist 77M Feb 13 2018 jre-8u161-linux-x64.tar.gz
-rw-r--r-- 1 idanyani assist 61M Jul 4 2015 jre-8u45-linux-x64.tar.gz
-rw-r--r-- 1 idanyani assist 1.9K Aug 25 2019 launch.jnlp
-rw-r--r-- 1 idanyani assist 898K Jan 8 2019 p7-Swift.pdf
-rw-r--r-- 1 idanyani assist 274K Aug 11 2015 p84-henning.pdf
-rw-r--r-- 1 idanyani assist 655K Jul 8 2018 pmu-tools-master.zip
-rw-r--r-- 1 idanyani assist 4.1K Apr 2 19:56 viewer.jnlp
[idanyani@csm ~/Downloads]$]]
```

## סגירת גישה לקובץ

```
#include <unistd.h>
int close(int fd);
```

- פעולה: סגורת את הקובץ המוצבע ע"י fd. לאחר הסגירה לא ניתן לגשת לקובץ דרך fd.
- פרמטרים:
  - fd – ה-FD המיועד לסגירה.
- ערך מוחזר: במקרה של הצלחה – 0. במקרה של כישלון – 1.

**שאלה:** close() מעדכנת את המצביע המתאים בטבלה ל-NULL, אבל לא בהכרח משחררת את ה-object file. **למה?**

יכול להיות שתהליכי/חווטים אחרים עדין מצביעים לאותו object file. יש מונה הסופר את כמות התהליכים המצביעים עליו בכל רגע נתון, ובכל close לעליו הוא מورد באחד. כאשר מתאפשר, ה-object file משוחרר.

## קריאה נתונים מקובץ

```
#include <unistd.h>
ssize_t read(int fd,
 void *buf,
 size_t count);
```

- **פעולה:** מנסה לקרוא עד count בתים מתוך הקובץ המקשר ל-fd לתוכן החיצן buf.
- מחוון הקובץ (ה-**seek pointer**) מקודם בכמות הבטים שנקראו, כך שבפעולה הגישה הבאה לקובץ (קריאה, כתיבה וכד') ניגש נתונים אחרים הנתונים שנקראו בפעולת הנוכחות.
- פעולה הקריאה עשויה לחסום את התהיליך (כלומר, להוציא אותו מהמתנה) עד שייהיו נתונים זמינים לקריאה, למשל עד שיגיעו נתונים מהדיםק.

משתמשים בו-ssize\_t, בנויגוד ל-t\_size, כי הוא יכול להחזיק גם ערכים שליליים. Read עלולה להחזיר -1 בכישלון

## קריאה נתונים מקובץ

### פרמטרים:

- fd – FD המקשר לקובץ ממנו מבקשים לקרוא.
- buf – מצביע לחיצן בו יאחסנו הנתונים שייקראו.
- count – מספר הבטים המבוקש.

### ערך מוחזר:

- במקרה של הצלחה – מספר הבטים שנקראו בפועל מהקובץ לתוך buf.
- ניתן שייקראו פחות מ-count בתים, למשל אם נותרו פחות מ-count בתים בקובץ ממנו קוראים.
- ניתן גם שלא יקראו בתים כלל, למשל אם מחוון הקובץ הגיע לסוף הקובץ (EOF).
- אם () read נקראה עם  $0 = \text{count}$ , יוחזר 0 ללא קריאה.
- במקרה של כישלון – (-1).

## כתיבה נתונים לקובץ

```
#include <unistd.h>
ssize_t write(int fd,
 const void *buf,
 size_t count);
```

- **פעולה:** מנסה לכתוב עד count בתים מתוך החוץ buf לקובץ המקשר ל-fd.
  - בדומה ל-**read()**, מחוון הקובץ מוקדם בכמות הבטים שנכתבו בפועל, והגישה הבאה לקובץ תהיה לנtones שאחרי אלו שנכתבו.
  - גם פועלות **write()** יכולה לחסום את התהיליך (כלומר, להוציא אותו מהמתנה), למשל עד שתתאפשר גישה לדיסק.

## כתיבה נתונים לקובץ

### פרמטרים:

- fd – FD המקשר לקובץ אליו מבקשים לכתוב.
- buf – מצביע לחיצן בו מאוחסנים הנתונים שייכתבו.
- count – מספר הבטים המבוקש לכתיבה.

### ערך מוחזר:

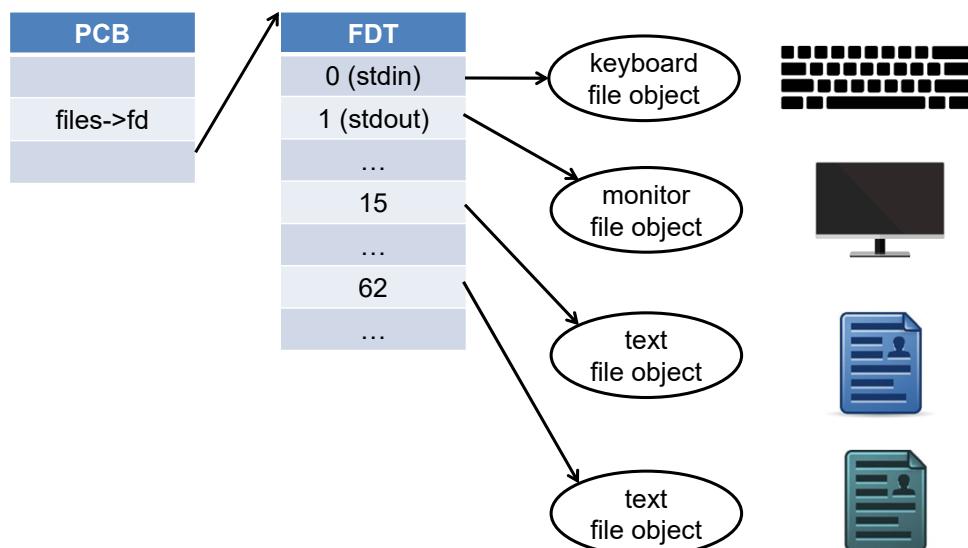
- במקרה של הצלחה – מספר הבטים שנכתב בפועל לקובץ מתוך `buf`.
- יתכן שייכתו פחות מ-`count` בתים, למשל אם אין מספיק מקום פנוי בדיסק.
- אם (`write` נקראת עם `0`, `count = 0`), יוחזר `0` ללא כתיבה.
- במקרה של כישלון – `(-1)`.

הפסקה

## Damn! Linux is so violent

```
root@terminal:~
root@terminal:~# love
-bash: love: not found
root@terminal:~# happiness
-bash: happiness: not found
root@terminal:~# peace
-bash: peace: not found
root@terminal:~# kill
-bash: you need to specify whom to kill
```

## file objects



## file objects

- קריית המערכת (`open`) מוסיפה כניסה חדשה **במקום הפנו** הראשון בטבלת FDT של התהילן.
- הכניסה החדשה מציבה על אובייקט מטיפוס `file`:

```
struct file {
 atomic_t f_count;
 ...
 loff_t f_pos;
 mode_t f_mode;
 ...
 struct file_operations* f_op;
};
```

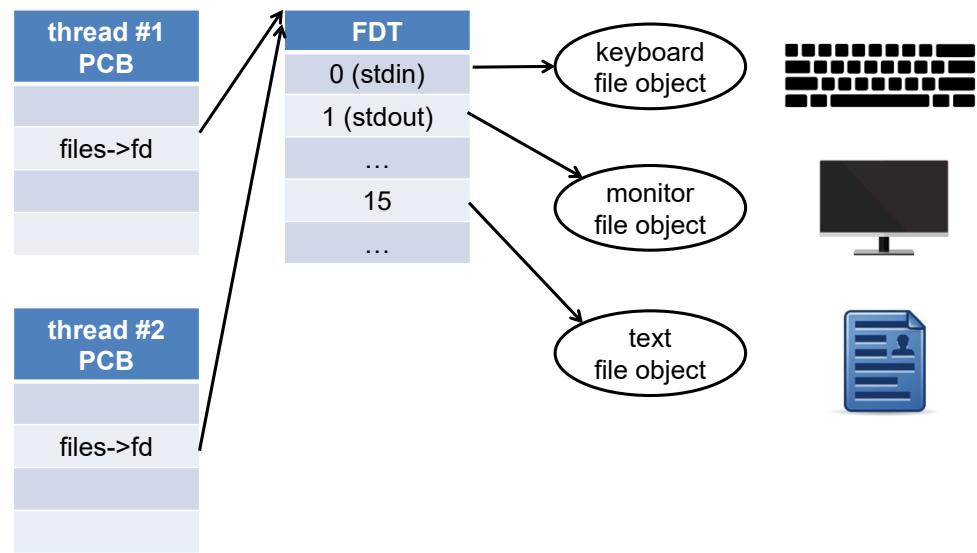
include/linux/fs.h  
include/asm-i386/fcntl.h

## file objects

- **f\_count** – סופר את מספר ההצלבות לאותו אובייקט.
- למשל: תהילכי אב ובן יציבו לאותו אובייקט לאחר `(fork())`.
- למשל: אותו תהילר יכול להציב פעמים לאותו אובייקט בעקבות (`dup()`).
- המונה משמש לשחרור האובייקט ב-`(close())` מהתהליך האחרון המציב.
- **f\_pos** – ה- `seek-pointer` – מצביע למקום הקריאה או הכתיבה הנוכחי'.
- **f\_mode** – שומר את הרשאות הקובץ, למשל האם הקובץ ניתן לקרוא/כתב.
- מערכת הפעלה תבדוק את שדה זה לפני ביצוע הפעולות `read`, `write`.
- **file\_operations** – מבנה המכיל מצביעים למימוש של הפעולות `open`, `read`, `write`, `lseek` ועוד רבות אחרות. נדבר על כך שוב בתרגול על מודולים ודריברים.

include/linux/fs.h  
include/asm-i386/fcntl.h

## שיתוף קלט/פלט בין חוטים



## שיתוף קלט/פלט בין חוטים

- חוטים של אותו תהיליך משתפים ביניהם את ה-FDT.
- מתאריו התהיליכים של כל החוטים מצביעים על אותו TDT.
- אם חוט אחד פותח קובץ גם החוט השני יוכל לגשת לקובץ זהה.
- אם חוט אחד סגור קובץ אז גם החוט השני לא יוכל לגשת אליו יותר.
- חוטים (ותהיליכים) המשתמשים ב-FD משותפים צריכים לתאמם את פעולות הגישה לקבצים על-מנת שלא לשבש זה את פעולות זהה.
- לינוקס מציע מגוון אפשרויות לנעילה של קבצים – מעבר לחומר הקורס.

אפשרויות לנעילת קבצים בלינוקס:

Advisory locking – תהיליך צריך לנעול קובץ בצורה מפורשת (כמו קטע קריטי).  
Mandatory locking – אם הקובץ נעול – כל תהיליך שמבצע open, read, write, "ינעל", גם אם לא ביצע נגילה מפורשת.

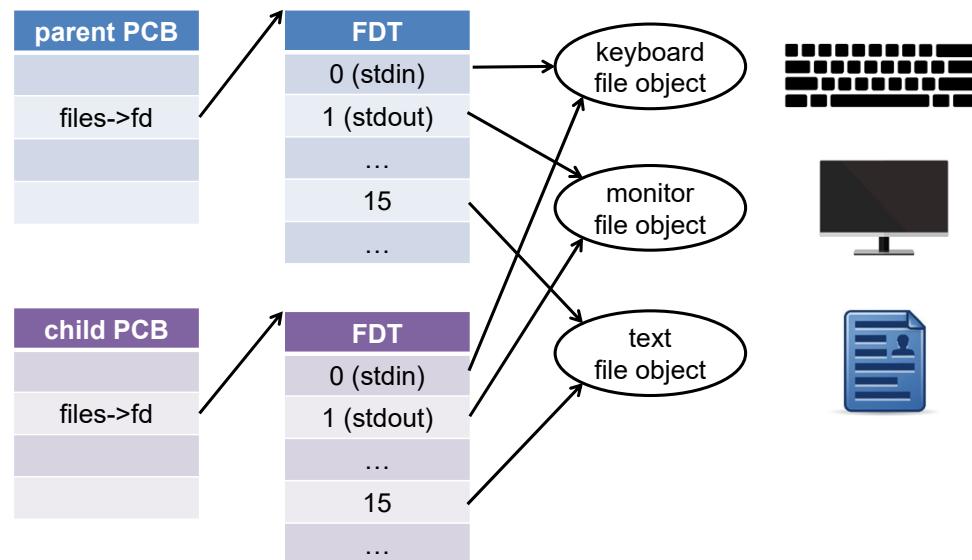
אפשר לנעול קובץ שלם או חלקים מהקובץ.  
leases – אם תהיליך שנעול את הקובץ לא משחרר אותו ולא מאיר את הlease תוך פרק זמן נתון – המנעול משחרר.

חשוב במקורה שתהיליך שנעול את הקובץ נפל.

שני טיפוסי מנעולים:

.flock FL\_LOCK  
.fcntl FL\_POSIX ע"י הפונקציה ()

## שיתוף קלט/פלט בין תהליכים



2 copies of FDT, but just one file object for each file.

## שיתוף קלט/פלט בין תהליכיים

- קריית המערכת () מעתיקה את ה-FDT לתהילך הבן,
- כל שינוי ב-FDT אצל האב/הבן לאחר ()fork לא נראה אצל השני.
- לדוגמה: אם תהילך הבן פותח קובץ חדש הוא לא נפתח אצל האבא.
- אבל ה-`objects file` (הקבצים הפתוחים) זהים אצל האב ואצל הבן.
  - שדות ב `file object` כמו `מחוון הקובץ`, יהיו זהים.
- לדוגמה: אם האב קורא 10 בתיים מהקובץ ולאחריו הבן קורא 3 בתיים, אז הבן יקרא את 3 הבתיים שהאחרי ה-10 של האב.
- שימושו לב: כל פתיחה של קובץ מייצרת `object file` חדש.
  - לדוגמה: אם אותו תהילך פותח פעמיים את אותו קובץ, אז הגרעין ישמור שני `file objects` שונים המצביעים לאזוריים שונים באותו הקובץ.

## שחרור file object

- **שאלה:** מי מבצע את שחרור הזיכרון של ?file object של ?
- מתי ניתן לשחררו?

- ייתכנו מצבים בהם תהליכי שונים מצביעים לאוטו file object, אך שחרור ה-object file יכול להתבצע רק לאחר ביצוע () close מכל התהליכי החולקים אותו ה-object file.
- נזכר - ל-object file יש מונה (f\_count) הסופר את כמות התהליכי המצביעים עליו בכל רגע נתון. המונה קטנה באחת עם כל פעולה () close על האובייקט.
- כאשר המונה מתאפס, ה-object file ישוחרר.

## שיתוף קלט/פלט לאחר exec()

shell code:

```
pid_t pid = fork();
if (pid == 0) {
 close(1);
 open("file.txt",
 O_CREAT ..., ...);
 char* args[] =
 {"date", NULL};
 execv(args[0],
 args);
} else {
 wait(NULL);
}
```

- פועלות execv ודומותיה אין משנה את ה-FDT של התהילר, למרות שהטהילר מאוחחל מחדש.
- כמובן, קבצים פתוחים אינם נסגרים.

- התוכנה הזאת שימושית להכוונת קלט/פלט של תכניות (input/output redirection).

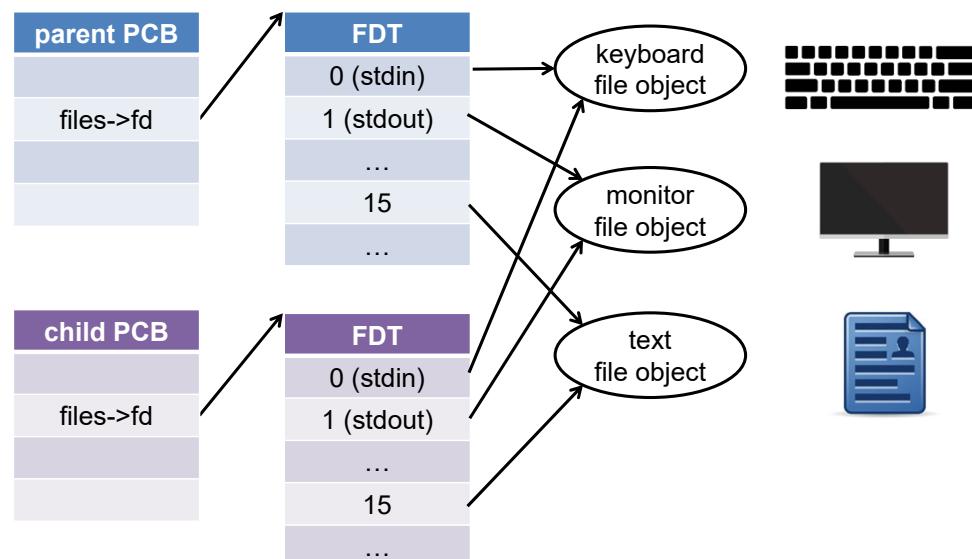
- למשל, ניתן לכתוב את התאריך והשעה הנוכחיים לקובץ במקומם לפלאט הסטנדרטי באמצעות:

>> date > file.txt

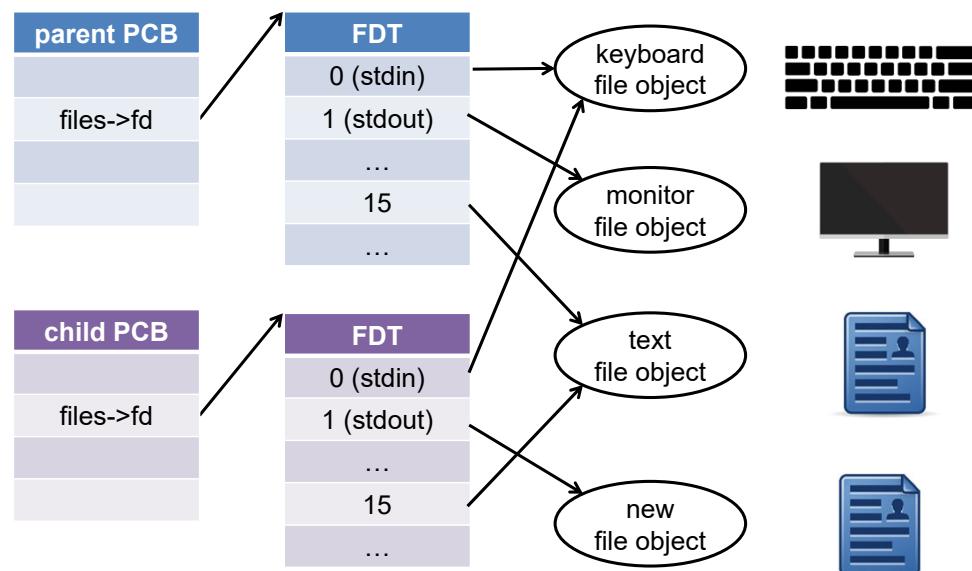
דוגמת הקוד בשקף זהה מדגימה מדוע בلينוקס החליטו להפריד יצרת תהילר חדש לשתי קריאות מערכת, fork+execv, כדי לאפשר לבצע מספר פקודות בין שתי קרייאות המערכת הללו.

שימוש לב שם execv היה מאפשר את טבלת הקבצים הפתוחים, לא היינו יכולים ממש הפניאת קלט/פלט בצורה הנ"ל.

## הכוונה קלט/פלט



## הכוונת קלט/פלט



## – מה רأינו עד עכšíו – Check Point

- דיברנו על דרך לספר לתהליכי על אירוע.
- באמצעות סיג널ים (signals).
- דיברנו כיצד מתבצע קלט/פלט בלינוקס.
  - באמצעות קבצים (files).
  - "Everything is a file".
- ועכšíו – נדבר על "קבצים" מיוחדים לתקשורת בין תהליכי:
  - Pipe – תקשורת בין תהליכים עם קשר משפחתי.
  - FIFO – תקשורת בין תהליכים כלשם.

## pipes בلينوكס



## pipes בلينוקס

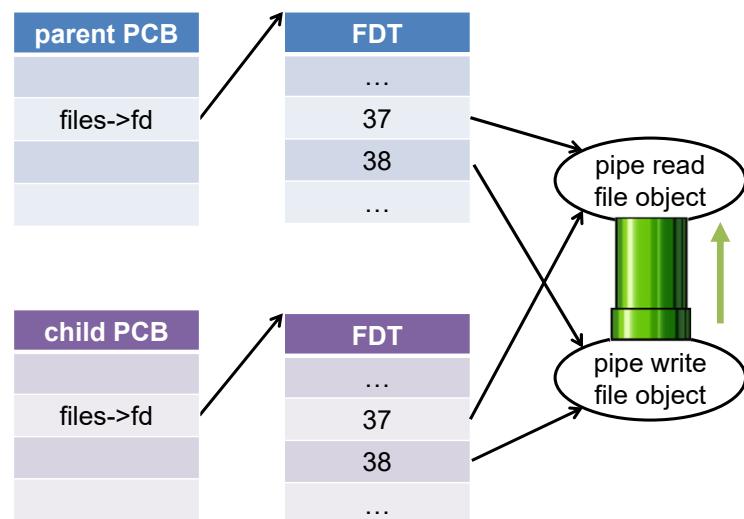
- עורך תקשורת חד-כיווני בסדר FIFO.
- pipes מאפשרים העברת מידע בין תהליכים.
- ניתן להשתמש ב-pipes גם לscanf'ון בין תהליכים.
- מערכת ההפעלה לינוקס מימושת sockets באמצעות "קבצים".
  - קיימים FD לכתיבה, ו-NFD לקרואיה.
  - קראיה וכ כתיבה כמו לקבצים וריגלים (ע"י קריאות המערכת read/write).
  - השימוש אינו צורך כל שטח דיסק, והוא מופיע בהיררכיה של מערכת הקבצים.
  - המידע שנכתב ל-socket נשמר בחוצצים (buffers) של מערכת ההפעלה.

## יצירת pipe חדש

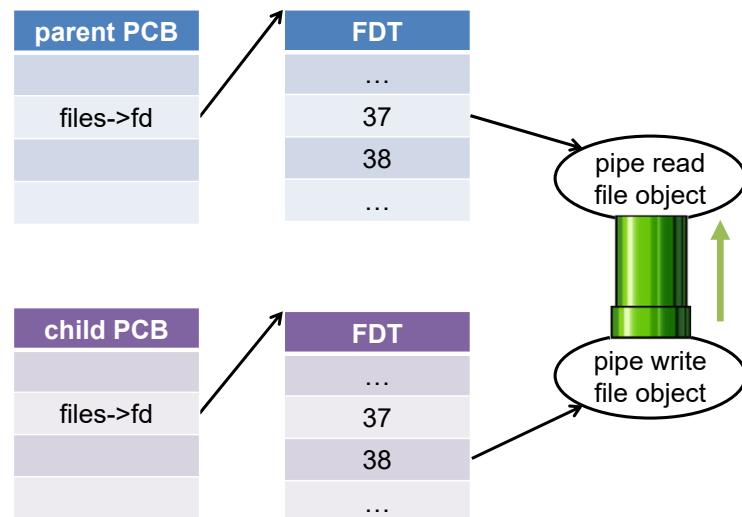
```
#include <unistd.h>
int pipe(int filedes[2]);
```

- פעולה: יוצרת סוקט חדש עם שני FD (שני קצנות הציגור) אחד לקריאה מה-pipe ואחד לכתיבה אליו.
- פרמטרים:
  - filedes – מערך בן שני תאים.
  - ב-[0] יאוחסן ה-FD **לקריאה** מה-pipe.
  - ב-[1] יאוחסן ה-FD **לכתיבה** מה-pipe.
  - הכניסות המוקצחות ל-pipe הן הראשונות הפנויות ב-FDT.
- ערך מוחזר: 0 בהצלחה ו-(1-) בכישלון.

## בלינוקו pipes



## בלינוקו pipes



כל תחיליך סגר את הקצה שלא רלוונטי אליו

## שיתוף pipes

- ה-exec'וק הנוצר הינו פרטיו לתהיליך ואינו נגיש לתהיליכים אחרים.
- **שיתוף exec'וק יבוצע בדרך אחת בלבד – בעזרה קשיי משפחה:**
  - תהיליך אב יוצר exec'וק – שתי כניסה חדשות נספנות ל-FDT שלו.
  - תהיליך האב יוצר תהיליך בן באמצעות ()fork – ה-FDT משוכפל לתהיליך הבן.
  - כעת לשני התהיליכים, האב והבן, יש גישה לה-exec'וק באמצעות ה-FD שלו.
- exec'וק הוא חד-כיווני ולכן האב והבן משחקים תפקידים שונים: האב קורא והבן כותב, או להפר.
- האב והבן צריכים לסגור את ה-FD השני שאינו בשימוש.
- לאחר סיום השימוש ב-exec'וק מצד כל התהיליכים (סגירת כל ה-FD)  
מפניים משאבי ה-exec'וק באופן אוטומטי.

## תכנות דוגמה – pipe

```
int main() {
 int my_pipe[2];
 char buff[6];
 pipe(my_pipe);

 if (fork() == 0) { // son
 close(my_pipe[0]);
 write(my_pipe[1], "Hello", 6);
 } else { // father
 close(my_pipe[1]);
 read(my_pipe[0], buff, 6);
 printf("Got from pipe: %s\n", buff);
 }
 return 0;
}
```

מה קורה אם האבא מתחיל לזרע לפני הבן?

מהו הפלט של התוכנית הנ"ל (בהתנזה שכל קריאות המערכת מצליחות?)

Answer #1: the father read() will block until the child will fill the buffer.

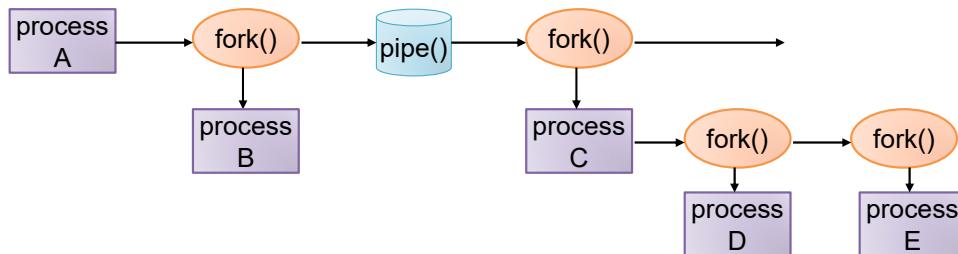
Answer #2: "Got from pipe: Hello"

Remember that the user should check the return values of these system call because they can fail.



## pipes בLinux

- למי מההתהליכים הבאים יש גישה ל-pipe שיצר תהליך A?



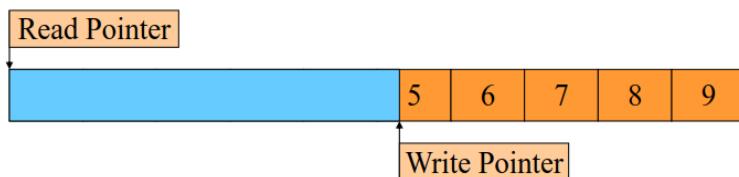
- תשובה: לכל התהליכים פרט ל-B.

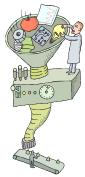
שאלה: מה קורה אם C סגור את קצה הכתיבה של ה-pipe לפני יצירת הבנים שלו? איז D וגם E לא יכולים לכתוב ל-pipe?

שאלה נוספת: מה קורה אם אחרי שכל התהליכים נוצרים A סגור את הקצה לקריאה אצלו? כל התהליכים האחרים עדין יכולים לקרוא! אבל A כבר לא.

## קריאה וכתיבה ל-pipe (1)

- פעולות קירה וכתיבה מתבצעות באמצעות `(read | write)()` על ה-`FDs` של ה-`pipe`.
- ניתן להסתכל על סדרם כמו על תור FIFO עם מצביע קירה יחיד (להוציאת נתונים) ומצביע כתיבה יחיד (להכנסת נתונים).
- כל קירה (מלל תחילך שהוא) מה-`pipe` מקדמת את מצביע הקירה. באופן דומה, כל כתיבה מקדמת את מצביע הכתיבה.





## קריאה וכתיבה ל-pipe (2)

### • קרייה מ-עוקם תחzier:

- את כמות הנתונים המבוקשת אם היא נמצאת ב-עוקם.
- פחתה מהכמות המבוקשת אם זו הכמות הזמין ב-עוקם בזמן הקריאה.
- 0 (EOF) כאשר כל write descriptors (write descriptors) ל-pipe וה-עוקם.
- **תחסום** את התהלייר אם יש כתובים (write descriptors) ל-pipe וה-עוקם.
- ריק. כאשר תבוצע כתיבה, יוחזרו הנתונים שנכתבו עד לכמות המבוקשת.

### • כתיבה ל-עוקם תבצע:

- אם אין קוראים (read descriptors) – הכתיבה תיכשל והתהלייר יקבל סיגナル בשם SIGPIPE (broken pipe error).
- טיפול בירית מחדל בסיגנל זה: הריגת התהלייר.
- ה-עוקם מוגבל בגודלו (כ-64KB). אם יש מספיק מקום פנוי ב-עוקם, תבוצע כתיבה של כל הכמות המבוקשת מיד.
- אם אין מספיק מקום פנוי ב-עוקם, הכתיבה **תחסום** את התהלייר עד שהקוראים האחרים יפנו מקום ונitin יהיה לכתוב את כל הכמות הדרושה.

Question to ask: How do we avoid conflicts between concurrent read and write between two ends of the pipe?

Answer: The kernel uses a lock in order to avoid race scenarios.

## למה צריך לסגור קצוות מיותרים?

```

int main() {
 int fd[2];
 int grade;

 pipe(fd);
 if (fork() == 0) { // teacher
 close(fd[0]);
 do {
 grade = get_random_between(0, 100);
 write(fd[1], (void*)&grade, sizeof(int));
 } while (grade != 0);

 } else { // student
 close(fd[1]);
 while (read(fd[0], (void*)&grade, sizeof(int)) > 0) {
 if (grade == 100) break;
 }
 printf("Grade = %d\n", grade);
 }
 return 0;
}

```

מה יקרה אם נסיר את השורה הזאת?

מה יקרה אם נסיר את השורה הזאת?

בדוגמה זו המורה מגറיל ציונים ושולח אותם לתלמיד. ברגע שהמורה מגיע לציון 0 – הוא מפסיק. ברגע שהתלמיד מגיע לציון 100 – הוא מפסיק.

תשובה #1: המורה ימשיך להגריל ציונים ולשלוח אותם, גם אם הסטודנט סיים. במקרה הטוב, זה יבזבז קצת זמן עד שהמורה יגריל 0 ויסיים בעצמו.

במקרה הרע, המורה ימלא את ה-EOF מבלי להגריל 0 ואז יתקע לנוכח.

תשובה #2: התלמיד ימשיך לקרוא ציונים ולעולם לא יקבל EOF, גם אם המורה סיים. במקרה הטוב, המורה יגריל 0 ויסיים, ואז הסטודנט יתקע לנוכח.



## קריאה המערכת (dup)

```
#include <unistd.h>
int dup(int oldfd);
int dup2(int oldfd, int newfd);
```

- **פעולה:** משכפלת את ה-FD שמספרו oldfd ל-FD אחר בטבלה.
- עברו dup: ה-FD החדש הינו ה-FD הפנוי **בעל ערך הנמור ביותר בטבלה**.
- עברו dup2: ה-FD החדש הינו newfd, לאחר סגירה, אם היה פתוח.
- לאחר פעולה מוצלחת, fd ו-FD החדש מצביעים לאותו file object.
- **פרמטרים:**
  - oldfd – ה-FD המועד להעתקה – חייב להיות פתוח לפני ההעתקה.
- **ערך מוחזר:**
  - בהצלחה, מוחזר ה-FD החדש.
  - בכישלון מוחזר (-1).

dup() is a short for duplicate.

1

שאלה

## הכוונת קלט/פלט באמצעות pipes

- ממשו בעזרת `dup` או `2>dup` קטע קצר המדמה את הפקודה הבאה ב-`:shell`

```
>> ls | more
```

- תרגום:

- על ה-`shell` להתחיל שני תהליכי המריצים את `ls` ו-`more`.
- יש לבצע redirection בין ה-`STDOUT` של תהליך `ls` ל-`STDIN` של תהליך `more` באמצעות מנגןן ה-`pipe`.

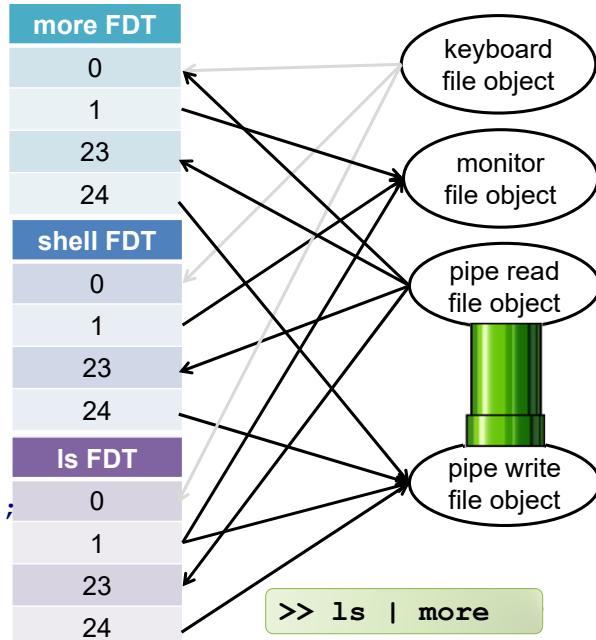
## הכוונת קלט/פלט באמצעות pipes - פתרון

```
// shell Code
int fd[2];
pipe(fd);

// שימו לב: תזמון הפעולות יכול
// להיות שונה מהסדר שבו
// הנפשות בשקף זה מוצגות
// (לדוגמא אם הבן השני
// התחל לזרץ לפני הראשון)

if (fork() == 0) {
 // second child
 dup2(fd[0], 0);
 close(fd[0]);
 close(fd[1]);
 execv("/bin/more", ...);
}
close(fd[0]);
close(fd[1]);
```

שימו לב: תזמון הפעולות יכול להיות שונה מהסדר שבו הנפשות בשקף זה מוצגות (לדוגמא אם הבן השני התחל לזרץ לפני הראשון)

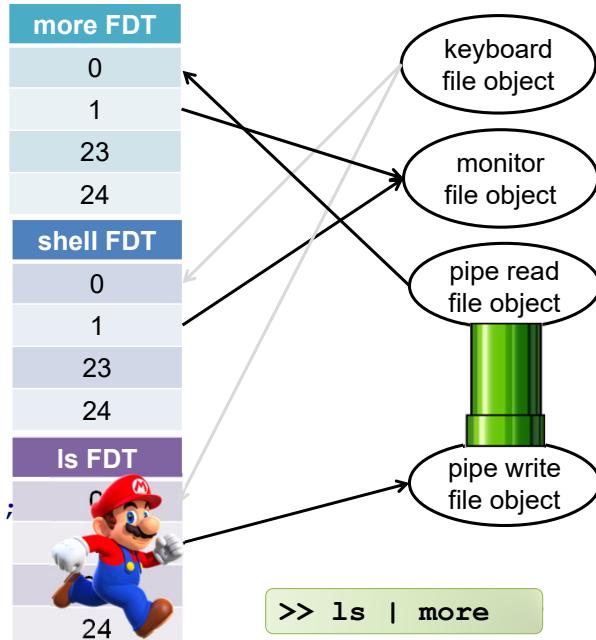


## הכוונת קלט/פלט באמצעות pipes - פתרון

```
// shell Code
int fd[2];
pipe(fd);

if (fork() == 0) {
 // first child
 dup2(fd[1], 1);
 close(fd[0]);
 close(fd[1]);
 execv("/bin/ls", ...);
}

if (fork() == 0) {
 // second child
 dup2(fd[0], 0);
 close(fd[0]);
 close(fd[1]);
 execv("/bin/more", ...);
}
close(fd[0]);
close(fd[1]);
```



## (named pipes) FIFOs

- FIFO הוא למעשה סדרון בעל "שם" במערכות הקבצים שדרכו יכולים כל התהיליכים במכונה לגשת אליו – **PIPE** "ציבורי".
- שם ה-FIFO הוא שם קובץ במערכות הקבצים, למשל: /home/yossi/myfifo .
- שימושו לרב: FIFO הוא קובץ **למרות שאיןנו נשמר על הדיסק**.
- pipes הם חסרי שם ולכך נקראים לפעמים FIFO anonymous .
- השימוש העיקרי של FIFO (או של כל אובייקט תקשורת בעל "שם") הוא כאשר תהיליכים רוצים לתקשר דרך ערוץ קבוע מראש **אבל** **שיינו בינהם קשרי משפחה**.
  - למשל, כאשר תהילכי לקוחות צריכים לתקשר עם תהיליך שרת.
- FIFO נוצר ע"י קריית המערכת (mkfifo), אשר מעניקה לו את שמו.

## קריאה המערכת (mkfifo)

```
#include <sys/types.h>
#include <sys/stat.h>
int mkfifo(const char *pathname,
 mode_t mode);
```

- פעולה: יוצרת FIFO המופיע במערכת הקבצים במסלול pathname והרשאות הגישה שלו הם mode.
- פרמטרים:
  - pathname – שם FIFO וגם המסלול לקובץ במערכת הקבצים.
  - mode – הרשאות הגישה ל-FIFO שנוצר. ניתן להכניס ערך 0777 (777 אוקטלי) כדי לאפשר הרשאות מלאות.
- ערך מוחזר: 0 בהצלחה, (-1) בכישלון.

## תקשורת באמצעות FIFO

- בניגוד ל-seqiom, O הינו ערוץ תקשורת דו-כיווני ובעל FD יחיד (ניתן לבצע הן קרייה והן כתיבה דרך אותו FD)
- ニגשים ל-FIFO באמצעות פקודות (chops). כתובים וקוראים באמצעות .read/write.
- תהליך שפותח את ה-FIFO לקריאה בלבד נחסם עד שתתהליך נוסף יפתח את ה-FIFO לכטיבה, וההפר.
- פתיחה ה-FIFO לכטיבה וקריאה (O\_RDWR) אינה חוסמת.
- חוקי הקריאה והכתיבה עובדים באופן דומה ל-seqiom.

## ניקוי שאריות

- כאובייקט בעל שם במערכת הקבצים, FIFO אינו מפונה אוטומטית לאחר שהמשתמש האחרון בו סגור את הקובץ.
- אין יש לפנותו בצורה מפורשת באמצעות פקודות או קריאות מערכת למחיקת קבצים (למשל, פקודת rm או קריית המערכת (unlink)).

## אז מה היה לנו היום:



- המטרה: לתקשר בין תהליכיים.
- signals (דיווח לתהיליך על אירוע).
- files (בין היתר בשבייל תקשורת בין תהליכיים).

Linux – Everything is a File

- .FDT
- .file objects
- .fork
- שיתוף אחרי

• קבצים מיוחדים לתקשורת בין תהליכיים:

- pipe
- FIFO
- sockets (לא דיברנו על זה היום).

• קריאות מערכות:

- כאלו שכבר הכרתם: open, close, write, read
- כאלו שלא הכרתם: signal, kill, pipe, mkfifo

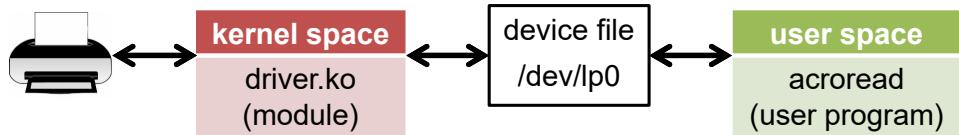
המטרה של השקף היא לרענן מה היה היום ולגרום לסטודנטים לשאול שאלות.

## תרגול 4

רצף האתחול (boot sequence) בלינוקס  
מודולים בלינוקס (loadable kernel modules)  
התקני תווים (character devices)  
דראיברים (device drivers)  
שימוש דראיברים באמצעות מודולים

## TL;DR

- **מודולים** בלינוקס הם תוספות קוד לגרעין שניתן לטעון בזמן ריצה.
- לא צריך להדר מחדש את הגרעין, לא צריך לאותחל את המחשב.
- מודולים משמשים להוספה פונקציונליות חדשה לגרעין, לדוגמה דרייברים להתקני חומרה שמחברים/מנתקים מהמערכת:
  - **התקני תווים** – מקלדת, עכבר, מדפסת, ...
  - **התקני בלוקים** – דיסק קשיח, disk-on-key, ...



## rzf האתחול בלינוקס

Linux Boot Sequence

Resources: <https://opensource.com/article/17/2/linux-boot-and-startup>

## אתחול המחשב

- כאשר המחשב כבוי, מערכת הפעלה, היישומים וכל הקבצים של המשתמש נשמרים על הדיסק.
- כאשר המשתמש מדליק את המחשב (באמצעות לחיצה על כפתור ON/POWER), מערכת הפעלה בדרך כלל לא נמצאת עדין בזיכרון ולכן צריך לטעון אותה מהדיסק.
- יש פה פרדוקס: כדי לטעון את מערכת הפעלה מהדיסק לזיכרון, צריך מערכת הפעלה שיוודעת לגשת להתקנים כמו הדיסק.
- הפרדוקס נפתר באמצעות תהליך אתחול הדרגתית של המחשב: סדרת רכיבי תוכנה שהמחשב מבצע כאשר המשתמש מדליק אותו.

## מה קורה לאחר הדלקת המחשב?



הSKUFL מתרגם את השלבים שקיורים כאשר המשתמש לוחץ על כפתור ההפעלה של המחשב ומדליק אותו.

## BIOS (Basic Input/Output System)

- התוכנה הראשונה שモפעלת לאחר הדלקת המחשב.
- קוד ה-BIOS צורב בזיכרון ייעודי בלוח האם ואז נטען לכתובות קבועה בזיכרון באופן אוטומטי ברגע הדלקת המחשב.

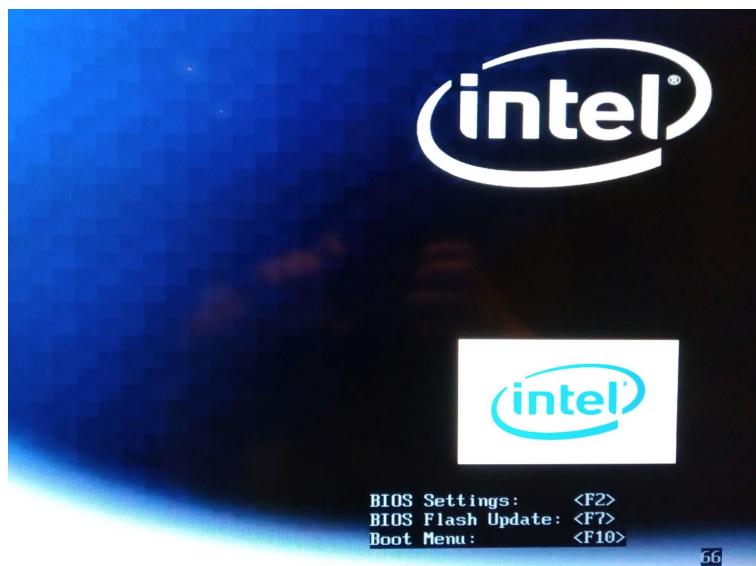
### • תפקידי ה-BIOS:

- לזרות את התקני החומרה המתחברים למערכת.
- לבדוק שההתקנים הבסיסיים ( מסך, מקלדת, ... ) פועלים בצורה תקינה.
- לעבור על רשימה מוגדרת מראש של התקנים, ולהיפש התקן המאפשר אתחול (bootable device).
- אם לא נמצא אף התקן צזה, ה-BIOS רושם הודעה שגיאה למסך ומפסיק את תהליך האיתחול.
- אם נמצא התקן צזה, ה-BIOS טוען את הסקטור הראשון של התקן למקום קבוע בזיכרון.

מי כותב את קוד ה-BIOS? יצרן לוח האם.

בעבר קוד ה-BIOS היה נשמר ברכיב איחסון לקריאה בלבד (read-only memory), אבל היום הוא בדרך כלל נשמר באמצעות מבועס flash כדי לאפשר עדכונים של הקוד. במחשבים מודרניים מגננון EFI-UEFI נמצא לפני BIOS נמצא בשימוש ומודעך על פניו BIOS.

## example BIOS screen



## MBR (master boot record)

- דיסק קשיח הוא התקן איחסון המאפשר לסקטוריים בגודל 512 בתים.
- קריאה/כתיבה מהדיסק נעשית בכפולות של סקטוריים.
- אם הדיסק הוא התקן המאפשר אתחול (bootable device), אז הסקטור הראשון בדיסק נקרא (MBR) master boot record.
- ה-MBR מכיל קוד אסמבלי בסיסי המשמש לטעינת קוד נוסף: מנהל האתחול (boot loader).
- יש מספר boot loaders נפוצים. מערכות לינוקס משתמשות בדרך כלל ב-GRUB.

למה לא לשמר את ה-  
boot loader ב-  
MBR?

תשובה: כמובן שהייה עדיף להכניס את כל ה-boot loader לסקטור אחד בדיסק (לו היה ניתן כדי לחסוך שלבים ברצף האתחול). אבל boot loaders מודרניים שוקלים יותר מ-512 בתים, ולכן לא ניתן לשמר אותם במלואם על סקטור יחיד בדיסק.

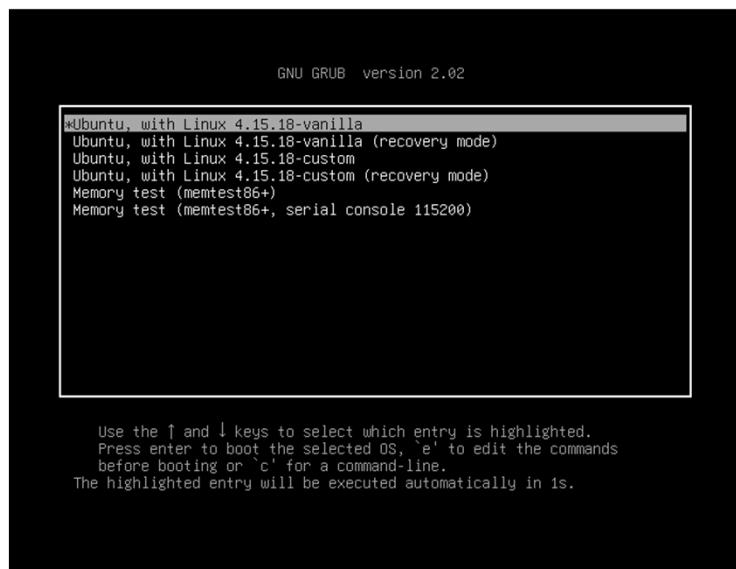
MBR is sometimes called the “stage 1” boot loader, and GRUB is called the “stage 2” boot loader.

## GRUB (GRand Unified Bootloader)

- בשקף הבא מופיע דוגמה של תפריט ה-GRUB: הוא מאפשר לבחור את גרעין מערכת הפעלה אשר יטען לזכרון.
- תפריט ה-GRUB שימושי כאשר מפתחים גרעין לינוקס חדש (כפי שתעשו בשיעורי הבית):
  - נניח כי עידכנתם את קוד הגרעין של לינוקס ושמרתם אותו בתמונה **vmlinuz-4.15.18-custom**.
  - לאחר מכן ניסיתם לטען את התמונה זהו לזכרון ומערכת הפעלה קרסה ☹  
לכלנו יש באגים לעממים...  
כעת תוכלו להפעיל מחדש את המחשב ולטען תמונה אחרת וטקינה, למשל **vmlinuz-4.15.18-vanilla**.
  - לאחר עליית מערכת הפעלה, תוכלו לנסות ולתמן את התמונה **vmlinuz-4.15.18-custom**.

שימוש לב: GRUB נדרש לדרייברים בסיסיים עבור גישה להתקני קלט/פלט כדי למשל:  
(1) לקבל הוראות מקלדת של המשטמש, (2) לטען מהדיסק את תמונה מערכת הפעלה  
(גרעין לינוקס או מערכת הפעלה אחרת).  
GRUB לא מממש בעצמו את הדרייברים האלו כי הם כבר ממומשים בקוד ה-BIOS, אשר  
נecessario להם בעצמו, למשל כדי לקרוא את ה-**MBR**.  
מספר הדרייברים ב-BIOS מצומצםיחסית למספר הדרייברים של לינוקס מספקת, ולכן ה-BIOS  
לא תמיד מזהה את כל הדיסקים במערכת.

## example GRUB menu

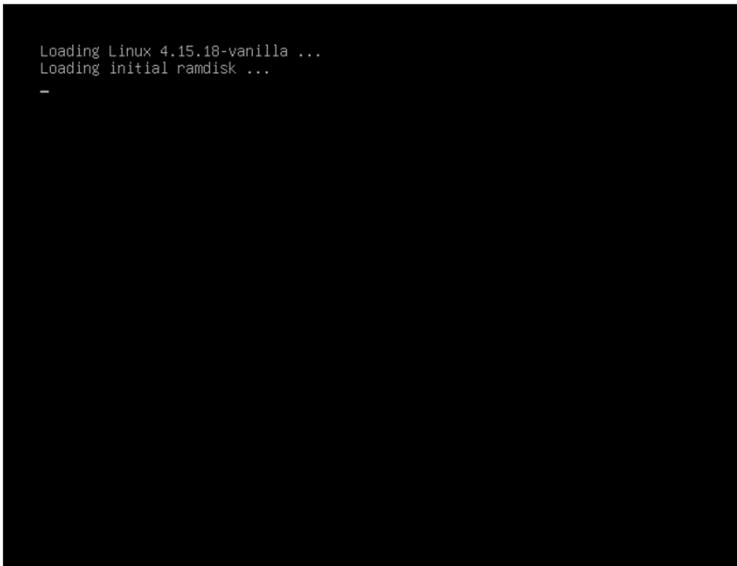


## טיענת גרעין לינוקס

- גם גרעין לינוקס נטען בשלבים:
  - .1 GRUB טוען לזכרון תמונה דחוסה של הגרעין, אשר נקראת בדרך כלל bzImage או vmlinuz, ואז מחלץ אותה.
  - .2 לצד תמונה הדחוסה של גרסת לינוקס 4.15 שוקלת בערך 8MB הראשונית: מערכת קבצים קטנה בשם initramfs או initrd.
    - initrd = initial RAM disk
    - initramfs = initial RAM file-system
  - .3 גרעין לינוקס מריצ' את התוכנית /init/ מתוך מערכת הקבצים הראשונית.
    - /init/ היא בדרך כלל סקרייפט shell.

מערכת הקבצים הראשונית שוקלת כמה עשרות MB בגרסה לינוקס 4.15.

## טיענת גרעין לינוקס



## מערכת הקבצים הראשונית

- מערכת הקבצים הראשונית מכילה את המודולים הנחוצים עבור גרעין לינוקס כדי לחפש ולטעון את מערכת הקבצים האמיתית.
  - לדוגמה: מודולים הממשים דרישות של הדיסק או כרטיס הרשת.
  - שימוש לב: מערכת הקבצים הראשונית לא מכילה את כל המודולים הקיימים, אלא רק את החינויים שבהם. שאר המודולים נמצאים במערכת הקבצים האמיתית, ולאחר שהיא תעלת גרעין לינוקס יוכל לטען גם אותם (במידת הצורך).
- .4 התוכנית `tzic` / טוענת את הדרישות הנחוצים ומרכיבה (`mounts`) את מערכת הקבצים האמיתית במקום המקורי של המערכת הקבצים הראשונית.
- .5 לאחר שעלה מערכת הקבצים האמיתית, גרעין לינוקס קורא לתוכנית `init` / `sb` / .
- היום `tzic` / `sbin` / הוא קישור לתוכנית `systemd` / `lib` / .

## >> dmesg -H

- פקודת `dmesg` מאפשרת לקרוא הודעות שהודפסו למסך בתהיליך האיתחול לאחר שהמערכת הופעלה:

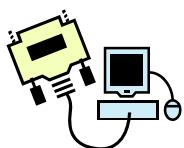
```
[Apr24 13:01] Linux version 4.15.18-custom (student@pc) (gcc version 7.5.0 (Ubuntu 7
[+0.000000] Command line: BOOT_IMAGE=/boot/vmlinuz-4.15.18-custom root=UUID=3c7375
[+0.000000] KERNEL supported cpus:
[+0.000000] Intel GenuineIntel
[+0.000000] AMD AuthenticAMD
[+0.000000] Centaur CentaurHauls
[+0.000000] Disabled fast string operations
[+0.000000] x86/fpu: Supporting XSAVE feature 0x001: 'x87 floating point registers'
[+0.000000] x86/fpu: Supporting XSAVE feature 0x002: 'SSE registers'
[+0.000000] x86/fpu: Supporting XSAVE feature 0x004: 'AVX registers'
[+0.000000] x86/fpu: Supporting XSAVE feature 0x008: 'MPX bounds registers'
[+0.000000] x86/fpu: Supporting XSAVE feature 0x010: 'MPX CSR'
[+0.000000] x86/fpu: xstate_offset[2]: 576, xstate_sizes[2]: 256
[+0.000000] x86/fpu: xstate_offset[3]: 832, xstate_sizes[3]: 64
[+0.000000] x86/fpu: xstate_offset[4]: 896, xstate_sizes[4]: 64
[+0.000000] x86/fpu: Enabled xstate features 0x1f, context size is 960 bytes, usin
[+0.000000] e820: BIOS-provided physical RAM map:
[+0.000000] BIOS-e820: [mem 0x0000000000000000-0x00000000009e7ff] usable
[+0.000000] BIOS-e820: [mem 0x0000000000009e800-0x00000000009ffff] reserved
[+0.000000] BIOS-e820: [mem 0x0000000000dc000-0x0000000000ffff] reserved
[+0.000000] BIOS-e820: [mem 0x0000000000100000-0x00000000bfecffff] usable
[+0.000000] BIOS-e820: [mem 0x000000000bfed0000-0x00000000bfefefff] ACPI data
[+0.000000] BIOS-e820: [mem 0x000000000bfef000-0x000000000bfefffff] ACPI NVS
...]
```

פקודת `dmesg` היא שימושית מכיוון שקשה לעקוב אחר ההודעות שהודפסו למסך בתהיליך האיתחול:  
 יש הרבה מאוד הודעות והן מופיעות בזמן קצר בגלל שהודעות חדשות שmag'יות "דוחפות"  
 אותן למעלה.

## מודולים בלינוקו

Loadable Kernel Modules

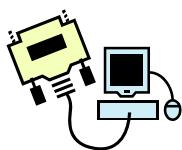
Read more: <https://www.tldp.org/LDP/lkmpg/2.6/html/x121.html>



## מודולים (modules)

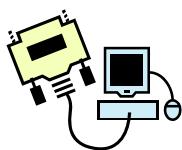
- **מודולים** מאפשרים להויסיף לגרעין לינוקס, בזמן ריצה, קטעי קוד חדשים.
- מודול הוא ספרייה משותפת (shared library) הננתענת (מקושרת) בזמן ריצה ופועלת במצב גרעין (כלומר  $\text{CPL}==0$ ).
- הפקודה **insmod** טעונה מודול חדש.
- הפקודה **rmmmod** פורקת מודול שנטען בעבר.
- הפקודה **lsmod** מציגה את רשימת המודולים הפעילים (כלומר, שנטענו ע"י המשתמש).

השם המלא של מודולים הוא: LKM = loadable kernel module.



## למה מושגים מודולים?

- מודולים משמשים בעיקר על מנת להויסיף תמייהה בהתקני חומרה (drivers) ע"י דרייברים (devices):
  - התקני תווים – מקלדת, עכבר, מדפסת, ...
  - התקני בלוקים – דיסק קשיח, disk-on-key, ...
  - התקני רשת – נראת בתרגולים הבאים.
- למעשה, מודולים רצים בהרשות גרעין ולכן הם יכולים להויסיף ולעדכן כל פונקציונליות של הגרעין:
  - להויסיף מערכות קבצים חדשות.
  - להויסיף קריאות מערכת חדשות ו/או לשנות קריאות מערכת קיימות.



## יתרונות השימוש במודולים

.1. ניתן להוסיף יכולות חדשות לגרעין מוביל לקמפל אותו וambil לאותחל (reboot) את המערכת.

.2. מודולים מפותחים בנפרד מהגרעין ← פחות שורות קוד בגרעין ← זמן קומpileציה קצר יותר.

- אין צורך לקמפל פונקציוניות מיותרת, למשל דרישים לחומרה שלא נמצאת ברשותנו.

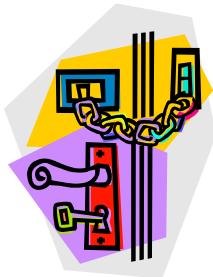
.3. הגרעין תופס פחות זכרון.

- המשמש טווען לזכור רק מודולים שהוא זקוק להם.
- ניתן לפרק מודולים שאינם בשימוש ולשחרר זיכרון.

.4. ניתן להוסיף בעתיד תמייהה בחומרה חדשה שעדיין לא קיימת.

## עניני אבטחה

- מודול רץ במצב גרעין ולכן יש לו גישה לכל מבני הנתונים בגרעין.
- חשוב להימנע מחורי אבטחה במודול:
  - למשל: הקפדה על אתחול משתנים, בדיקת תקינות קלט משתמש כדי למנוע overrun buffer וכוכי.
- כמו כן, לינוקס מגבילה טוונת מודולים למשתמשים מורשים (root) בלבד.



למרות שאבטחה אינה אחד מנושאי הקורס, לא ניתן לדון בכתיבה מודולים בלי להתייחס  
לנושא.

## דוגמת קוד: מודול ראשון

```
#include <linux/module.h> // always required
#include <linux/kernel.h> // for printk

int init_module(void) {
 printk("Hello World!\n");
 return 0;
}

void cleanup_module(void) {
 printk("Goodbye cruel world!\n");
}
```

נקראת בטעינת המודול

האם אפשר לקרוא ל-  
printf במקום ?printf

תשובה: לא. הפונקציה printk היא הגרסה של גרעין לינוקס לפונקציה printf והיא ממומנת בקורס גרעין באופן עצמאי ובלתי תלוי ב-fprintf.  
 קוד של מודול (כמו כל קוד גרעין) אינו יכול להשתמש בפונקציות של הספרייה libc, בפרט הפונקציה printf.  
 הסיבה לכך היא שהגרעין נטען לזכרון עוד לפני שספריית libc נגישה, ולכן הוא אינו מחובר לספרייה libc ברגע שתוכניות משתמש בקריאה מערכתי, ואילו קוד גרעין לא זוקן כמו כן, פונקציות הספרייה libc משמשות בקריאה מערכתי, ואילו קוד גרעין לא זוקן לקרוא מערכתי כי הוא רץ בהרשאות גבוהות.

## בנייה המודול

- קובץ makefile לדוגמה:

```
obj-m += hello.o
all:
 make -C /lib/modules/$(shell uname -r)/build
 M=$(PWD) modules
clean:
 make -C /lib/modules/$(shell uname -r)/build
 M=$(PWD) clean
```

- חיבת להיות התאמה מלאה בין גרסת לינוקס עליה נבנה המודול לבין גרסת לינוקס בה הוא רץ.

- חוסר תאימות עשוי לגרום לביעיות בזמן ריצה.

## טעינת המודול

```
>> make
>> sudo insmod ./hello.ko
Hello world!
>> lsmod
Module Size Used by
hello 868 0 (unused)
...
...
>> sudo rmmod hello
Goodbye cruel world!
```



שימוש לברק משתמשים מורשיים (root) יכולם להשתמש בפקודה זו.  
modprobe טעונה מודולים מתוך הנתיב . /lib/modules  
modprobe is the intelligent version of insmod. insmod simply adds a module  
where modprobe looks for any dependency (if that module is dependent on any  
other module) and loads them.

## העברת פרמטרים למודול

- הגדרת הפרמטרים בקוד המודול:

```
#include <linux/moduleparam.h>
int iValue=0; // 0 is the default value
char *sValue;
module_param(iValue, int, S_IRUGO);
module_param(sValue, charp, S_IRUGO);
```

- העברת פרמטרים בטיענת המודול:

```
>> insmod ./params.ko iValue=3
sValue="hello"
```

ניתן גם להעביר מערך בתור פרמטר:

```
int iArray[4];
module_param_array(iArray, int, &len, S_IRUGO);
```

טריק להעברת פרמטר אופציונילי: מערך בגודל 0–1.



## העברה פרמטרים למודול

- מודולים יכולים לקבל פרמטרים מהמשתמש בזמן תעינתם.
- הפרמטרים מוגדרים בקוד באמצעות המאקרו `module_param`.
- המאקרו צריך להופיע מחוץ לפונקציה. בד"כ ממוקם בתחילת הקוד.
- פרמטר ראשון – המשטנה שיכיל את הפרמטר, פרמטר שני – סוג הפרמטר, פרמטר שלישי – הרשות גישה לקובץ המתאים ב-`sysfs` (לא רלוונטי כרגע).
- יש להגדיר לכל פרמטר ערך ברירת מחדל.
- סוגי פרמטרים לדוגמה: `int`, `bool`, `charp`, `short`, `int`, `byte`.
- ניתן להשתמש במאקרו `MODULE_PARM_DESC` כדי להוסיף תיאור לפרמטר.
- כל ניהול אוטומטיים יכולים לקרוא את התיאור (`modinfo` גם עם `-o`).

## גישה לנוטוני גרעין

- למודול יש גישה למבנה הנוטונים של הגרעין במידה והוא מצרף את הקבצים המתאימים (ע"י `#include`).

```
#include <linux/sched.h>

int init_module(void)
{
 printk("The process is \"%s\" (pid %d)\n",
 current->comm, current->pid);
 return 0;
}
```

היכן שדה השומר את שם התוכנית המתבצעת. מה שם התוכנית שיודפס במקרה זה?

תשובה: התהיליך שטוען את המודול הוא התהיליך שמרייך את התוכנית `insmod` (זהו תהיליך בן של ה-`shell`).  
לכן שם התוכנית שיודפס הוא `insmod`. דוגמת הרצה:  
> `insmod ./hello.ko`  
The process is "insmod" (pid 21676)

## התקני תווים Character Devices

---

## התקנים (drivers) ודריברים (devices)

### • התקנים:

- מיוצגים ע"י קבצים מיוחדים בנתיב `/dev` במערכת הקבצים.
- המשמש עובד מול התקן באמצעות **המשק הסטנדרטי** לעובדת מול קבצים – קריונות המערכת (`read()`, `write()`, `open()`).
- לדוגמה: כדי להשתמש במדפסת יש לפתח את הקובץ `0&/dev` ואז לכתוב אליו את הטקסט להדפסה.

### • דריבר (מנהל התקן):

פועל בהרשאות גרעין וממפה את קריונות המערכת הללו לפעולות ספציפיות להתקן.

- לדוגמה: הדריבר של המדפסת "מדובר" עם המדפסת בפקודות ספציפיות עבורה (אייפה להדפס על הניר, מת מסתימת שורה של הדפסה, ...).
- דריבר הוא שכבת תוכנה החוצצת בין התקן לבין האפליקציה כדי לספק אבטחה לפעולות התקן הספציפי.

עקרונות העיצוב של התקנים: הפרדה בין "מכניזם" לבין "מדיניות".

מכניזם – מה אפשר לעשות (איזה יכולות מספקים)  
מדיניות – איך עושים (איך מנצלים את יכולות)

abilitat תוכנה היא קלה יותר לשימוש כחלקים שונים בתכנית מմמשים את התפקידים השונים.

דוגמה למנגנון החלפת הקשר. דוגמה למדיניות: מדיניות זימון תהליכיים (...RR, FCFS, SJF, ...) ועד דוגמה למנגנון דפודוף. דוגמה למדיניות: מדיניות החלפת דפים (...FIFO, LRU, ...).

באופן דומה, דריבר מטפל רק במנגנון (למשל, איך להפעיל כונן דיסקטים), והמדיניות מסופקת ברמות גבוהות יותר של מערכת הפעלה (מי יכול לגשת לכונן, מי רשאי לעשות לו `mount` וכן הלאה). אם אפשר, נעדיף לאפשר שינוי מדיניות ברמת `user mode`.

כשכתבם דריבר, מומלץ לשמר על גמישות – לטפל רק בגישה לחומרה, בלי להטיל אילוצים נוספים על המדיניות – זה נשאר לאפליקציה.

במקרים מסוימים קיימת אפשרות לכתוב דריברים שרצים ב-`user mode`, ואין מוקומפלים עם קוד גרעין. לאណון בתרגול בדריברים מסווג זה.

## התקני תווים ובלוקים

### התקן בלוקים

#### (block devices)

- התקן שנייתן לגשת אליו רק בCAFOLOTOT של בלוק (למשל 512 בתים).
- לרוב משמשים לאחסון מידע. לדוגמה: דיסק קשיח, דיסק נשלף (disk on key).
- התקן בלוקים מאפשר גישה אקרואית למידע שבו.
- LINOKOS מוסיף שכבה נוספת מהתקני בלוקים גם בתים בודדים.

### התקן תווים

#### (character devices)

- התקן שנייגשים אליו כאלו רצף של בתים.
- לרוב משמשים להעברת מידע. לדוגמה: מסך, מקלדת.
- בדרך כלל ניתן לגשת לתקן תווים רק באופן סדרתי (ולא אקרואי).

מבחןת המשמש, התקני תווים והתקני בלוקים עשויים לתפקד באופן דומה. ההבדל ביניהם בא לידי ביטוי במשמעות הפנימי בין מערכת הפעלה לבין התקן.

הסבר נוסף: (מתוך- unix-system )

Block devices (also called block special files) usually behave a lot like ordinary files: they are an array of bytes, and the value that is read at a given location is the value that was last written there. Data from block device can be cached in memory and read back from cache; writes can be buffered. Block devices are normally seekable (i.e. there is a notion of position inside the file which the application can change). The name "block device" comes from the fact that the corresponding hardware typically reads and writes a whole block at a time (e.g. a sector on a hard disk).

Character devices (also called character special files) behave like pipes, serial ports, etc. Writing or reading to them is an immediate action. What the driver does with the data is its own business. Writing a byte to a character device might cause it to be displayed on screen, output on a serial port, converted into a sound, ... Reading a byte from a device might cause the serial port to wait for input, might return a random byte (/dev/urandom), ... The name "character device" comes from the fact that each character is handled individually.

ביקפדייה האנגלית נתונים הגדרה מעט שונה : [https://en.wikipedia.org/wiki/Device\\_file](https://en.wikipedia.org/wiki/Device_file)

*Character special files* or *character devices* provide unbuffered, direct access to the hardware device. They do not necessarily allow programs to read or write single characters at a time; that is up to the device in question. The character device for a hard disk, for example, will normally require that all reads and writes are aligned to block boundaries and most certainly will not allow reading a single byte.

Character devices are sometimes known as *raw devices* to avoid the confusion surrounding the fact that a character device for a piece of block-based hardware will typically require programs to read and write aligned blocks.

*Block special files* or *block devices* provide buffered access to hardware devices, and provide some abstraction from their specifics.<sup>[5]</sup> Unlike character devices, block devices will always allow the programmer to read or write a block of any size (including single characters/bytes) and any alignment. The downside is that because block devices are buffered, the programmer does not know how long it will take before written data is passed from the kernel's buffers to the actual device, or indeed in what order two separate writes will arrive at the physical device; additionally, if the same hardware exposes both character and block devices, there is a risk of data corruption due to clients using the character device being unaware of changes made in the buffers of the block device.

Most systems create both block and character devices to represent hardware like hard disks. FreeBSD and Linux notably do not; the former has removed support for block devices,<sup>[6]</sup> while the latter creates only block devices. In Linux, to get a character device for a disk one must use the "raw" driver, though one can get the same effect as opening a character device by opening the block device with the Linux-specific O\_DIRECT flag.

## התקני תווים

- התקן תווים מאופיין ע"י שני מספרים:
- מספר ראשי (major number)** – מזזה את הדרייבר הקשור להתקן.
- מספר שני (minor number)** – מזזה את התקן הספציפי הקשור לאוטו דרייבר (יכולם להיות מספר התקנים, למשל מספר עכברים, המנוהלים ע"י אותו דרייבר).

```
>> ls -l /dev
crw-rw-rw- 1 root root 1, 3 Aug 31 10:33 null
crw----- 1 root root 10, 1 May 12 10:33 psaux
crw----- 1 root tty 4, 1 May 12 10:33 ttys1
crw-rw-rw- 1 root root 1, 5 Aug 31 10:33 zero
```

התקני תווים מסומנים ע"י תוו c בעמודה הראשונה

למשל, התקנים null ו-zero שניהם מנוהלים ע"י הדרייבר המסומן במספר ראשי 1. המספר המשני מאפשר לדרייבר לבדוק בהבחין ביניהם.

אבחן נספთ: מספר ראשי משמש את קריית המערכת (open לזריהו הדרייבר של קובץ ההתקן. מספר שני משמש את הפונקציות של הדרייבר בלבד.

## התקני תווים פיקטיביים

- לינוקס מספקת גם התקני תווים פיקטיביים שאינם קשורים לחומרה אמיתית, כולם בעלי מספר ראשי 1.

| <b>write()</b>                                 | <b>read()</b>                                          | <b>device file</b>                          |
|------------------------------------------------|--------------------------------------------------------|---------------------------------------------|
| מצילחה ולא עושה דבר<br>(הميدע שנשלח נזק)       | מחזירה מיד EOF<br>(end of file)                        | /dev/null                                   |
| מחזירה מיד ENOSPC<br>(no space left on device) | מחזירה רצף אפסים<br>באורך המבוקש                       | /dev/zero                                   |
| כותבת לרצף הבטים האקראי                        | מחזירה רצף בתים<br>אקראי שנוצר בזמן ריצה<br>מתוך "רעש" | /dev/random<br>/dev/urandom<br>/dev/arandom |

## שאלות לווידוא הבנה

- מה מבצעת הפקודה?

```
>> some_program > /dev/null
```

- מרייצה את התוכנית `some_program` בחזית ומשתיקה את כל הדפסות שהיא שולץ הפלט הסטנדרטי.
- כלומר התוכנית לא תדפיס למסך (אלא אם כן יש הדפסות ל-`STDERR`).

- מה מבצעת הפקודה?

```
>> cat /dev/zero > file.txt
```

- מייצרת קובץ אינסופי באורך מלא באפסים.  
הפעולה לא תעצור עד שנשלח סיגナル הריגה עם `CTRL+C`.

 shell utility

## יצירת התקן חדש

**mknod** <NAME> <TYPE> <MAJOR> <MINOR>

- פעולה: יוצרת קובץ התקן חדש.
- הפעולה דורשת הרשאות root.

### פרמטרים:

- NAME – שם הקובץ החדש שיוצג את התקן.
  - TYPE – סוג התקן (c – התקן תווים, b – התקן בלוקים).
  - MAJOR – המספר הראשי של הדרייבר המפעיל את התקן.
  - MINOR – המספר המשני של התקן (מספר בין 0 ל-255).
- התוכנית mknod ממומנת באמצעות קריית המערכת (sudo).

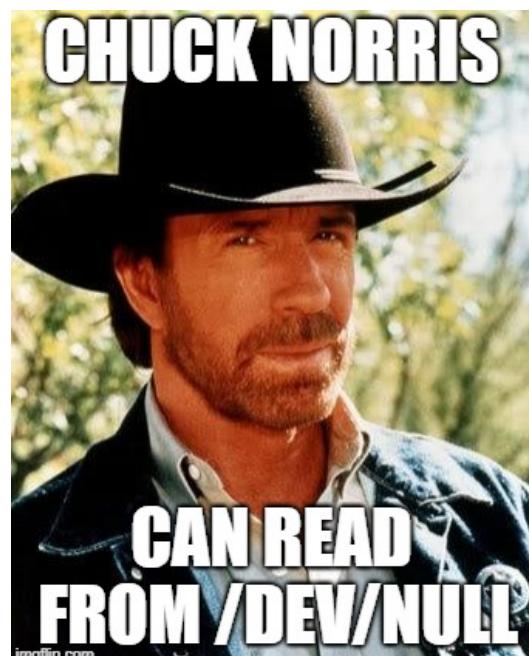
בפועל, משתמשי לינוקס לא באמת צרכים ליצור קובץ לכל התקן המחבר למערכת כי יש תהיליך מיוחד (ליתר דיוק DIMON) העושה זאת עבורם – udev. udev הוא תהיליך במרחב המשתמש אשר מזהה באופן דינמי התקנים המחברים למערכת וază יוצר עבורם קבצים תחת התיקייה /dev/.  
לקראיה נוספת: <https://en.wikipedia.org/wiki/Udev>

## דוגמה: ייצרת קובץ התקן חדש

```
>> mknod /dev/myDev c 254 0
```

- הפקודה תיצור קובץ בשם `/dev/myDev`, המיצג התקן תווים, המנוהל ע"י הדריבר הרשום עם מספר ראשי 254. המספר המשני של התקן הוא 0.
  - התקנים חדשים נוצרים כברירת מחדל עם הרשות כתיבה ליווצר ההתקן (לרוב root) וההרשאות קריאה לשאר המשתמשים.
  - במידת הצורך ניתן לשנות את הרשות אלה לאחר ייצרת התקן.
- ניתן להסיר התקן באמצעות דומה למחיקת קובץ רגיל:
- ```
>> rm /dev/myDev
```

הפרק



דрайיברים (מנהל התקנים)

Device Drivers

Read more: <https://www.tldp.org/LDP/lkmpg/2.6/html/x569.html>

תזכורת: file descriptors

- תזכורת: קריית המערכת (`open`) מקצת כניסה חדשה חדשה במקום הפנו הראישון ב-FDT של התהילר.
- הכניסה מצביעה על אובייקט מטיפוס `struct file`:

```
struct file {
    ...
    loff_t f_pos;
    ...
    void *private_data;
    ...
    struct file_operations* *f_op;
};
```

מצביע למקום הקריאה/הכתיבה הנוכחי

שדה המאותחל ל-NULL
ומאפשר לדרייבר לשמר
מידע נוסף (אם הוא צריך)

זה למעשה
הדריבר

הטיפוס `struct file_operations` מוגדר ב-<linux/fs.h>
משתנים מטיפוס `*f_op, fops, ...` נקראים בדרך כלל `struct file_operations`

פעולות על התקנים

- מערכת ההפעלה מגדרה אוסף **פעולות** שנייה לבצע על קבצים באמצעות קריאות מערכת ... `read()`, `write()`
- בפרט זהו גם אוסף הפעולות שנייה לבצע על התקן תווים.

- כל קובץ פתוח מציביע לבניה נתונים מסווג **file_operations**, שהוא מערך של מצביים לפונקציות הממשות את אותן הפעולות.
- `op_f` הוא השדה המצביע לבניה זה ב-`file struct`.
- מצביע `NULL` מייצג פונקציה לא ממומשת, או מיימוש ברירת מחדל.

מי מימוש את
הפעולות הללו?

- גישה מונחית עצמים:
- הקובץ הוא האובייקט.
- המתודות של האובייקט מוגדרות ע"י אוסף הfonקציות בס-`sops`.

תשובה: הדרייבר (בדרכו כלל נטען כמודול בזמן הריצה).

שלבי הפונקציה sys_read()

```
ssize_t sys_read(unsigned int fd,
                 char * buf, size_t count) {
    ...
    struct file * file = fget(fd);
    if (!file)
        return -EBADF;
    if (!(file->f_mode & FMODE_READ))
        return -EINVAL;
    if (file->f_op && file->f_op->read != NULL)
        return file->f_op->read(file, buf,
                                   count, &file->f_pos);
    ...
}
```

לאחר בדיקות תקינות הארגומנטים, מגיעים לחלק המרכז: הפעלת המתודה (read) שספקה ע"י הדריבר.
הקוד המופיע כאן שונה מהקוד המקורי לצורך פשטות ההדגמה.

שדות חשובים ב-file operations

- **open** – מצביע לפונקציה לפתיחת התקן.

- אם מואתחל ל-NULL, פעולה ()open תמיד תצליח.

```
int (*open) (struct inode *, struct file *);
```

- **release** – מצביע לפונקציה לשחרור התקן.

file object קריית המערכת ()close לא גוררת בהכרח קריאה ל-()release. אם ה-object משותף (למשל, לאחר ()fork), תבוצע קריאה ל-()release רק לאחר שכל העותקים של התקן נסגרו.

- כמו במקרה של ()open, ניתן לאתחל את הפונקציה ל-NULL.

```
int (*release) (struct inode *, struct file *);
```

- **flush** – מצביע לפונקציה לנקיי החוצצים וכתיבת המידע בהם ישירות לתקן.

모פעלת כל פעם שתהילר סוג העתק של התקן מסוים. במידה ומואתחל ל-NULL, מערכת ההפעלה לא תבצע את הפעולה.

```
int (*flush) (struct file *);
```

שימוש לב: החתימות של המתודות ב-file operations שונות מהחתימות של קריאות המערכת המקבילות.

שדות חשובים ב-file operations

- **read** – מצביע לפונקציה לקרוא מההתקן.

• במידה ומאותחל ל-NULL, קריאת המערכת `read` תחזיר `-EINVAL`.

```
ssize_t (*read) (struct file *, char *,
                   size_t, loff_t *);
```

- **write** – מצביע לפונקציה לכתיבה להתקן.

• במידה ומאותחל ל-NULL, קריאת המערכת `write` תחזיר `-EINVAL`.

```
ssize_t (*write) (struct file *, const char *,
                   size_t, loff_t *);
```

- **lseek** – מצביע לפונקציה לשינוי המיקום הנוכחי בקובץ.

• ישפייע על פעולות `read/write`.

• מוחזר את המיקום החדש בקובץ.

```
loff_t (*lseek) (struct file *, loff_t, int);
```

שדות חשובים ב-file operations

- **ioctl** – משמש להעברת פקודות ייחודיות לתקן. במידה ומאותחל .EINVAL, קריית המערכת ioctl תחזיר NULL-7.

```
int (*ioctl) (struct inode *, struct file *,  
unsigned int cmd_id, unsigned long arg);
```



קריאה המערכת (`ioctl()`)

```
int ioctl(int fd, int cmd, ...);
```

- פעולה: מאפשרת פקודות בקרה ייחודיות להתקן.
- יש להשתמש באפשרות זאת רק אם לא ניתן לספק מענה הולם במסגרת הפונקציות הקיימות. לדוגמה: פקודה `ejec` לכונן הדיסקים.
- פרמטרים:
 - `fd` – מתאר קובץ (שהתקבל כתוצאה של `(open)`).
 - `cmd` – מספר סידורי של פקודה (יועבר כמו שהוא לפונקציה המממשת).
 - ... – פרמטר אופציוני (המהדר לא יבודק אם הוא העובר או לא).
 - הפרמטר האופציוני עשוי להיות ערך שלם או מצביע. לאחר וההדר לא מבצע כל בדיקה, הפונקציה המממשת את ioctl צריכה לוודא כי הערך שהמשמש העביר הוא ערך חוקי המתאים להגדרת הפונקציה.
 - בדרך כלל מתייחסים לפרמטר האופציוני כ-`char *argp`.
- ערך חוזר: תלוי בכותב הדריבר (ערך 1 - מסמן שגיאה).

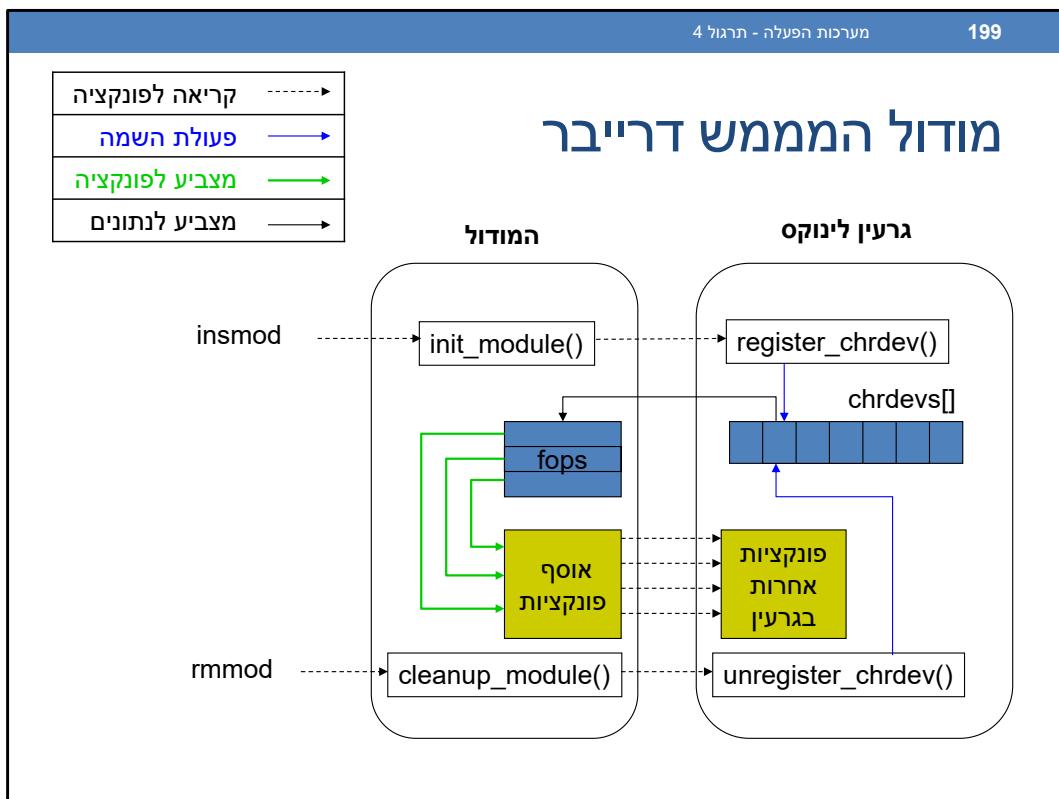
בעוד פעולות קלט/פלט הן סטנדרטיות ויש להן משקל משותף, לכל התקן חומרה עשויים להיות מאפיינים ייחודיים רק לו.

הפונקציה ioctl מהויה משק כללי להעברת פקודות בקרה להתקן חומרה. קריאת המערכת ioctl תגרום להפעלת הפונקציה ioctl המוגדרת עבור הקובץ (במידה ולא מוגדרת, יוחזר ערך שגיאה 1 - עם errno=EINVAL).

שימוש דרייברים באמצעות מודולים

דראיברים בתור מודולים

- ניתן לבנות דרייבר בתור **מודול** (כלומר, בנפרד משאר הגרעין) ואז "לחבר" אותו בזמן ריצה בשעת הצורך.
- המודול ירשום את הדרייבר במהלך הטעינה שלו.
 - כלומר הפונקציה (`init_module()`) תקרא לפונקציה (`register_chrdev()`).
- **שימוש לב:** רישום דרייבר רק מקשר בין דרייבר לבין מספר ראשי. אין קישור ישיר בין דרייבר להתקן.
- המודול ימחק את הרישום בזמן הפריקה שלו.
 - כלומר הפונקציה (`cleanup_module()`) תקרא לפונקציה (`unregister_chrdev()`).



(ההפרדה בתרשימים מתייחסת לקוד של המודול לעומת קוד גרעין קיימ. שימו לב שגם המודול נתען לאזרור ההזכרון של הגרעין) התרשימים הנ"ל מייצג עקרונות כללים העובודה עם מודולים, אבל אנחנו נטרח בתקני תווים.

השימוש בדראיברים מבוסס על גישה מונחית אובייקטיבים (דומה ל-ADT במת"מ). כל אובייקט (במקרה זה דרייבר) יודע איך לבצע פעולות שונות, ואובייקטים שונים יכולים לבצע את אותה פעולה בדרךים שונות. כאשר טוענים את המודול של הדרייבר, הוא רושם את עצמו במערכת באמצעות הפונקציה `register_chrdev`. הוא מעביר למערכת מבנה המכיל מצביעים לאוסף של פונקציות. פונקציות אלה ממופות לאוסף פונקציות המוכר ע"י הגרעין (כגון `read`, `open`, ...). התחן הפיזי מנהל ע"י אוסף הפונקציות המוגדרות עבור הדרייבר שלו. למשל, כאשר נבצע `open` על הקובץ המייצג את התחן, תבוצע הפונקציה `open` הייחודית לדרייבר. אם רוצים, ניתן אפילו לקבל גמישות גדולה יותר: הדרייבר יכול להציג פונקציות שונות בהתאם לתקנים פיזיים שונים המנוהלים על-ידי אותו דרייבר).

מערך הדריברים הרשומים

	name	fops
0		
1		
2		
3		
4		
...		

- המערך [`chrdevs`] שומר את כל הדריברים הרשומים כרגע במערכת.
- כל תא במערך מכיל לפחות שני שדות:
 - שם הדריבר.
 - מצבייע ל-`.file_operations`.
- האינדקס לערך הוא המספר הראשי של הדריבר.

On Linux 4.15, `chrdevs` is a hash table and not an array.
See the code in `fs/char_dev.c` .

רישום דרייבר חדש

```
int register_chrdev(unsigned int major,  
                    const char *name,  
                    struct file_operations *fops);
```

- פעולה: רישמת דרייבר חדש להתקני תווים ומקצה לו מספר ראשי.
- מוסיפה רישום שלו לקובץ `/proc/devices`.
- מוסיפה רישום שלו במערך הדריברים `.chrdevs`.
- פרמטרים:
 - `major` – המספר הראשי אותו רוצים להקנות לדרייבר.
 - `name` – שם הדרייבר כפי שיופיע ב-`/proc/devices`.
 - `fops` – מערך של מצביעי פונקציות, הממשים את פעולות התקן.
- ערך מוחזר: במקרה של הצלחה יחזיר ערך 0 או חיובי, אחרת -1.

פונקציה גרעין, לא קריית מערכת!

שימוש לב: הפונקציה `register_chrdev` (מוגדרת ב-`fs.h`) היא בעלת שם מטעה, כי היא משמשת לרישום דרייבר.

הקצת מספר ראשי

- לינוקס תומך ב-512 מספרים ראשיים.
- חלק מהמספרים הראשיים מוקצים באופן סטטי להתקנים נפוצים.
- הקצאה סטטית של מספרים ראשיים יכולה להיות בעייתית אם שני התקנים שונים יבקשו אותו מספר ראשי.
- ניתן ועדיין לבצע **הקצתה דינמית** של מספר ראשי ע"י העברת ערך **0** עבור הפקט `major`.
- מערכות הפעלה תחפש מספר ראשי פנוי החל מ-512 ומעלה.
- המספר הראשי שנבחר יהיה ערך החזרה של `(register_chrdev()`.
- במקרה של הקצאה סטטית ערך החזרה יהיה 0.
- ניתן לראות את המספר שהוקצה גם ב-`/proc/devices`.

ניתן לראות איזה מספרים ראשיים מוקצים באופן סטטי בקובץ `txt` בקובץ `Documentation/devices.txt` (בעז הראשי של קוד הגרעין).

>> cat /proc/devices

Character devices:

```
1 mem
4 /dev/vc/0
4 tty
...
...
```

Block devices:

```
7 loop
8 sd
9 md
...
...
```

• /proc/devices הוא קובץ

המפרט את כל הדריברים הרשומים כרגע במערכת.

• שימושו ליב שקיים גם מערך נוסף, של דרייברים עبور התקני בלוקים, ולכן ישכו כפליות של מספרי major.

• התקנים עצם נשמרים בתור קבצים בתיקיה /dev,

ומקשרים לדרייברים באמצעות המספר הראשי שלהם.

• זכרו כי המשמש יוצר קובץ התקן עם מספר ראשי מסוים באמצעות פקודת mknod.

Read more at:

<https://superuser.com/questions/1253047/why-dev-and-proc-devices-have-totally-different-outputs-in-linux>

<https://unix.stackexchange.com/questions/216684/what-is-the-relationship-similarity-and-differences-between-proc-devices-and>

פונקציה גרעין, לא קריית מערכת!

הסרת דרייבר רשום

```
int unregister_chrdev(unsigned int major,  
const char *name);
```

- **פעולה:** מסירה את הדרייבר ע"י שחרור המספר הראשי שהוקצה לו.
- אינה מוחקת את קובץ התקן מ-/dev/.
- **פרמטרים:**
 - major – המספר הראשי של הדרייבר אותו רוצים להסיר.
 - name – שם הדרייבר, כפי שמופיע ב-/proc/devices/.
- **ערך מוחזר:** במקרה של הצלחה יוחזר ערך 0 או חיובי, אחרת 1.

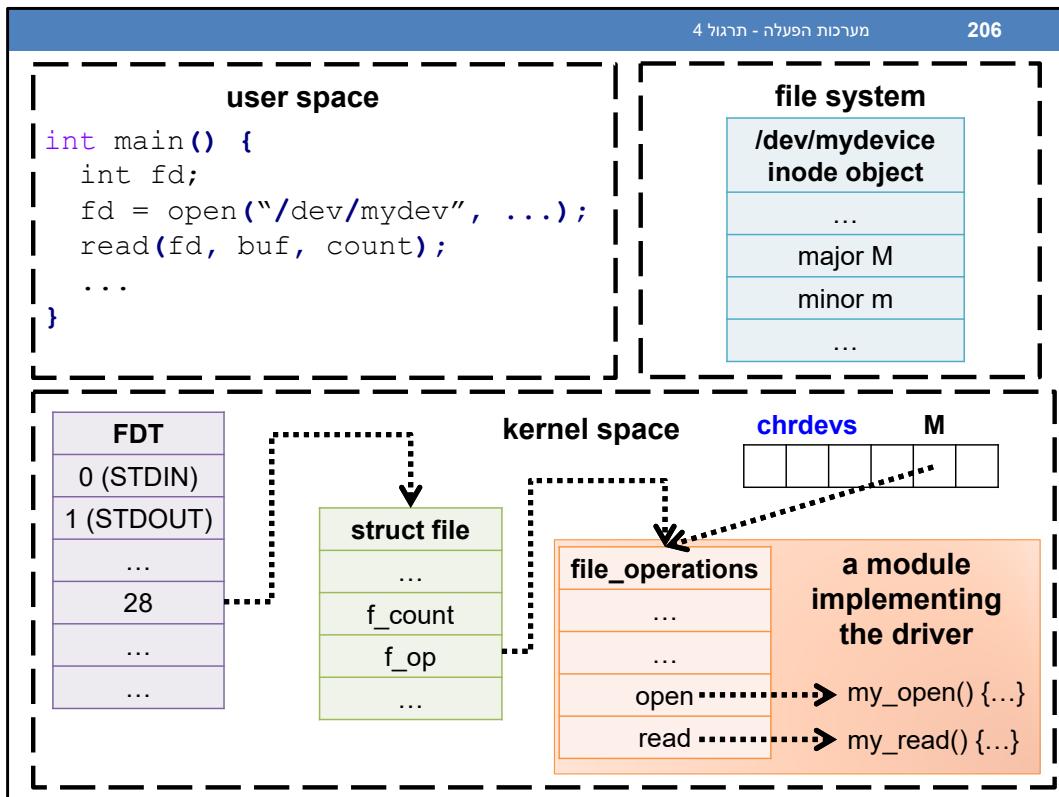
אם המודול של הדרייבר הוסר (למשל באמצעות `rmmod`) ללא שחרור המספר הראשי, עלולים לקבל שגיאות במערכת (למשל פניה לכתובת לא חוקית בזמן שפותחים את קובץ התקן).

אם עדין יש התקנים העשויים שימוש במספר הראשי, אנחנו עלולים לקבל שגיאות בזמן שימוש בהם. במקרה הטוב, אין דרייבר שלו ניתן להפנות את הבקשות ואז המערכת תקroles. במקרה הרע, דרייבר אחר תפס את המספר הראשי והוא יטפל בבקשתות.

rzf השלבים בקריאה המערכת (open)

- לסיום, נלמד כיצד מומשת קריאת המערכת (open).
- במילים אחרות: מה שלבי הפונקציה (open) ?
- נקוב אחרי הדוגמה הבאה:

```
int main() {  
    int fd;  
    fd = open("/dev/mydev", ...);  
    read(fd, buf, count);  
    ...  
}
```



The `file_operations` struct is **contained** within the module implementing the driver.

שלבי הפעונקציה (`sys_open()`)

- .1 ניגשת למערכת הקבצים וקוראת את המספר הראשי `M` והמשני `m` של התחן.
- .2 בודקת כי הדרייבר של התחן רשום: במידה ו- `NULL==chrdevs[M]` – `ENODEV` תוחזר שגיאה.
- .3 מתחילה `object` `file` חדש (נקרא לו `filp`) ומקצת כניסה חדשה ב-FDT שמצויה עליו.
- .4 מצביעה את `op_f` \rightarrow `filp` לפעונקציות של הדרייבר `op_f` \rightarrow `chrdevs[M]`.
- .5 במידה והפעונקציה (`open`) \rightarrow `op_f` \rightarrow `chrdevs[M]` אינה `NULL`, קוראת לה ומעבירה את `filp` כפרמטר.
- הפעונקציה (`open`) של הדרייבר تعدכן את `op_f` \rightarrow `filp` כדי להגדיר התנהגות ספציפית להתחן – נראה בהמשך התרגול.
- .6 לבסוף, מחזירה את ה-FD (הכניסה שהוקצתה עבור הקובץ ב-FDT).

קריאה המערכת (`open` מבצעת פעולות נוספות הקשורות בחיפוש הקובץ מעבר להפעלת ה-`(open` של הדרייבר.
פרטים נוספים בתרגול על מערכות קבצים.

זיהוי התקן ע"י דרייבר

- שימושו לב לחתימה של `(open)` :

```
int (*open) (struct inode *, struct file *);
```

- הפונקציה מקבלת כפרמטר את ה-`inode` המיצג את התקן.
- כל קובץ פתוח מיוצג ע"י מבנה נתוניים בשם `inode` (פרטים נוספים בתרגול 13). בפרט, גם לתקן תווים יש `inode` המציג אותו.
- השדה `v_rdev` ב-`inode` מכיל את **המספר הראשי והמשני** של התקן.
 - המאקרו `MAJOR(inode->i_rdev)` מחזיר את המספר הראשי.
 - המאקרו `MINOR(inode->i_rdev)` מחזיר את המספר המשני.

דוגמת קוד – מודול המממש דרייבר (1)

```
#include "linux/module.h"

int major = 0; /* will hold the driver major number */

struct file_operations my_fops = {
    .open=    my_open,
    .release=   my_release,
    .read=    my_read,
    .write=   my_write,
    .ioctl=   my_ioctl,
};

struct file_operations my_fops2 = {
    .open=    my_open,
    .release=   my_release2,
    .read=    my_read2,
    .write=   my_write2,
    .ioctl=   my_ioctl,
};
```

דוגמת קוד – מודול המממש דרייבר (2)

```
int init_module( void ) {
    major = register_chrdev(major, "my_module", &my_fops);

    if( major < 0 ) {
        printk(KERN_WARNING "Bad dynamic major\n");
        return major;
    }
    //do_init();
    return 0;
}

void cleanup_module( void ) {
    unregister_chrdev(major, "my_module");
    //do_clean_up();
}
```

דוגמת קוד – מודול המממש דרייבר (3)

```
int my_open(struct inode *inode, struct file *filp) {
    filp->private_data = allocate_private_data();
    if( filp->f_mode & FMODE_READ )
        // handle read opening
    if( filp->f_mode & FMODE_WRITE )
        // handle write opening

    if (MINOR(inode->i_rdev) == 2)
        filp->f_op = &my_fops2;
    return 0;
}
```

() my_open מחליפה את fops כתלות במספר המינורי.
שימוש לב: fops2 לא רשום בתור דרייבר במערך chrdevs
אבל הוא עדין נגיש מתוך קוד המודול.

דוגמת קוד – מודול המממש דרייבר (4)

```
ssize_t my_read(struct file
*filp, char *buf, size_t count,
loff_t *f_pos) {
    // custom implementation 1
}

ssize_t my_write(struct file
*filp, const char *buf, size_t
count, loff_t *f_pos) {
    // custom implementation 1
}

int my_release(struct inode
*inode, struct file *filp) {
    // custom implementation 1
}

ssize_t my_read2(struct file
*filp, char *buf, size_t count,
loff_t *f_pos) {
    // custom implementation 2
}

ssize_t my_write2(struct file
*filp, const char *buf, size_t
count, loff_t *f_pos) {
    // custom implementation 2
}

int my_release2(struct inode
*inode, struct file *filp) {
    // custom implementation 2
}
```

דוגמת קוד – מודול המממש דרייבר (5)

```
int my_ioctl(struct inode *inode,
             struct file *filp,
             unsigned int cmd,
             unsigned long arg) {

    switch( cmd ) {
        case MY_OP1:
            //handle op1;
            break;

        case MY_OP2:
            //handle op2;
            break;

        default:
            return -ENOTTY;
    }
    return 0;
}
```

תרגול 5

מבוא לזמן תהליכים

זמן תהליכי זמן אמת בלבד

זמן תהליכים רגילים בלבד

TL;DR

- **זמן התהליכים (scheduler)** הוא הרכיב במערכת ההפעלה שאחראי על בחירת התהליך הבא שירוץ על המעבד.

- אנחנו Learned, בתרור דוגמה, את אלגוריתם הזמן שמיון של לינוקס.

זמן התהליכים בלינוקס

CFS = completely fair scheduler

אלגוריתם זמן שמיון של תהליכי זמן רגילים

SCHED_FIFO, SCHED_RR

אלגוריתם זמן שמיון של תהליכי זמן אמיתי

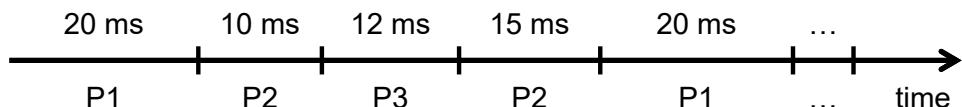
- אבל לפני שנלמד דוגמה "אמיתית" ומורכבת, נרצה להבין את הגישות הבסיסיות בזמן התהליכים כדי לפתח אינטואיציה.

מבוא לזמןון תהליכיים

החומר בפרק זה מבוסס על פרק 7 (Scheduling: Introduction) מהספר OSTEP.

הגדרת הבעיה

- במערכות מחשבים אמיטיות, השמייקה קצרה מדי: בכל רגע מחייב לרווח יותר תהליכיים מסווגים מעבדים שיכולים להריץ אותם.
- אין ברירה: מערכת הפעלה צריכה לחלק את זמן המעבדים בין התהליכים.



- איך כדאי לבחור, בכל נקודת זמן, את התהליך הבא שיוצג על המעבד? וכמה זמן למתן לתהליך זה?
- אין תשובה "נכונה". נדרש למצוא פשרה בין מספר דרישות הנדסיות: הוגנות, אינטראקטיביות ויעילות.

יעילות, אינטראקטיביות, והוגנות

- **יעילות** – תהליכיים מסתויים מהר ככל הניתן.
 - בדרך כלל יעילות מוגדרת באמצעות מدد זמן ההמתנה הממוצע.
- **אינטראקטיביות** – תהליכיים מגיבים מהר לפעולות O/I.
 - לא היינו רוצים ששיחת טלפון תהיה מוקוטעת כי תהליך חיפוש קבצים רץ ברקע.
- **הוגנות** – אין הגדרה חד-משמעות, אלא יש כמה גישות.
 - למשל: תהליך שהגיע ראשון, ירוץ ראשון על המעבד.
 - למשל: כל התהליכים מקבלים את אותו זמן מעבד.
- שלושת הדרישות סותרות אחת את השניה, ולכן יש למצוא איזון ביןיהן.

5 הנחות על העומס

- נרצה לפתח אלגוריתם זימון בסיסי.
- לשם כך נניח 5 הנחות מפשטות ולא ריאליות על העומס (workload) במערכת:
 - .1. כל התהיליכים רצויים למשך אותו זמן.
 - .2. כל התהיליכים מגיעים באותו זמן ($t=0$).
 - .3. אם תחילת התחליל לרוץ, אז הוא ירוץ עד לסיוםו ללא הפסוקות.
 - .4. התהיליכים משתמשים רק במעבד ולא מבצעים O/I.
 - .5. זמן הריצה של כל התהיליכים ידוע מראש.
- בהמשך נסיר לאט את ההנחות האלו כדי לפתח אלגוריתם "אמיתי".

הנחה מובלעת (נדבר עליה שוב בהמשך): כל תחילת רץ על ליבת מעבד אחת, כלומר התהיליכים לא מקבילים.

מדדי ביצועים (מטריקות)

- כדי להשוות בין אלגוריתמי זימון שונים באופן כמוותי, נגידיר מספר מדדי ביצועים.
- המדד הראשון יהיה: "זמן התגובה הממוצע".
- לכל תחילה נגידיר את "זמן התגובה":
$$\text{responseTime} = \text{terminationTime} - \text{arrivalTime}$$
- מכיוון שיש הרבה תהליכיים, נמציע את זמן התגובה על פני כולם.
- כרגע, תחת ההנחה שהגדרכנו, כל התהליכיים מגיעים בזמן 0.
- ◀ **זמן התגובה = זמן הסיום.**

1. כל התהליכים רצים למשך אותו זמן.
2. כל התהליכים מגיעם באותו זמן ($t=0$).
3. אם תחילן התחליל לרוץ, אז הוא ירוץ עד לסיוםו ללא הפסקות.
4. התהליכים משתמשים רק בזיכרון ולא מבצעים O/I.
5. זמן הריצה של כל התהליכים ידוע מראש.

אלגוריתם FCFS

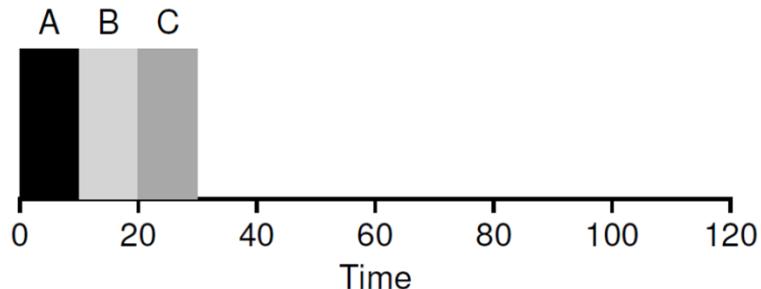
- $FCFS = \text{first come, first served}$
- נקרא גם: FIFO = first in, first out
- אופן פעולה: תור הוגן.

- לדוגמה:
- נניח כי 3 תהליכים A,B,C הגיעו בזמן $t=0$ למערכת.
- נניח ש-A הגיע טיפה לפני B, שהגיע טיפה לפני C.
- לבסוף נניח כי זמן הריצה של כל אחד מהתהליכים הוא 10 שניות.
- איך נראה הזמן? מה זמן התגובה הממוצע?

1. כל התהליכים רצים למשך אותו זמן.
2. כל התהליכים מגיעים באותו זמן ($t=0$).).
3. אם תחילן התחליל לרץ, אז הוא ירץ עד סיוםו ללא הפסקות.
4. התהליכים משתמשים רק במעבד ולא מבצעים I/O.
5. זמן הריצה של כל התהליכים ידוע מראש.

דוגמה FCFS

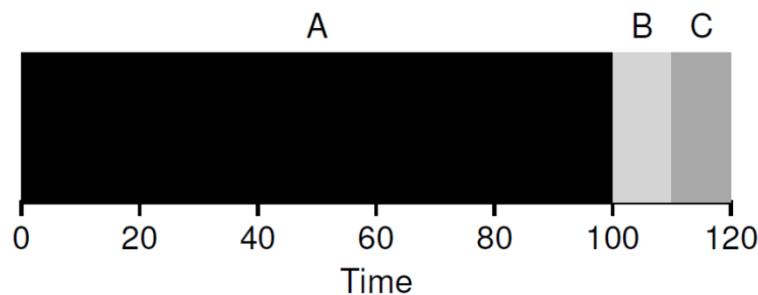
$$\text{averageResponseTime} = (10 + 20 + 30) / 3 = 20$$



- כתעת נסיר את הנחה 1 ("כל התהליכים רצים למשך אותו זמן").
← לכל תחילן זמן ריצה משלו.
- תוכלו לחושב על דוגמה שבה FCFS אינו יעיל?

אפקט השיירה (convoy effect)

$$\text{averageResponseTime} = (100 + 110 + 120) / 3 = 110$$



- אלגוריתם FCFS עלול לסבול מ"אפקט השיירה": מצב שבו תהליכי אחד ארוך מעכבר הרבה תהליכי קצרים.

1. כל התהליכים רציטם למשך אותו זמן.
2. כל התהליכים מגיעים באותו זמן ($t=0$).
3. אם תחילן התחליל לרץ, אז הוא ירץ עד לסיוםו ללא הפסיקות.
4. התהליכים משתמשים רק במעבד ולא מבצעים I/O.
5. זמן הריצה של כל התהליכים ידוע מראש.

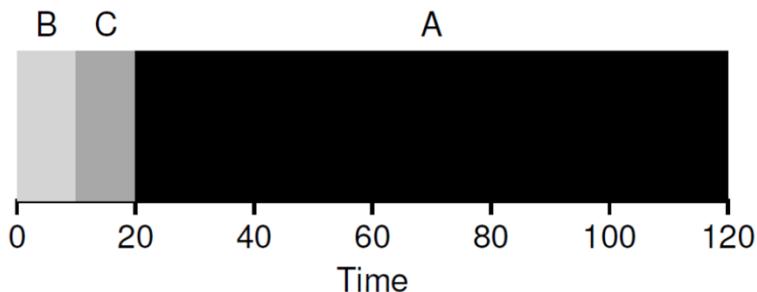
אלגוריתם SJF

SJF = shortest job first •

• אופן פועלה: השם אומר הכל.

• בדוגמה الأخيرة קיבל:

$$\text{averageResponseTime} = (10 + 20 + 120) / 3 = 50$$



1. כל התהליכיים רציטים למשך אותו זמן.
2. כל התהליכיים מגיעים באותו זמן ($t=0$).
3. אם תחילן התחליל רוץ, אז הוא ירוץ עד סיוםו ללא הפסקות.
4. התהליכיים משמשים רק בעקבות מאובנים/O/I.
5. זמן הריצה של כל התהליכיים ידוע מראש.

אופטימליות של SJF

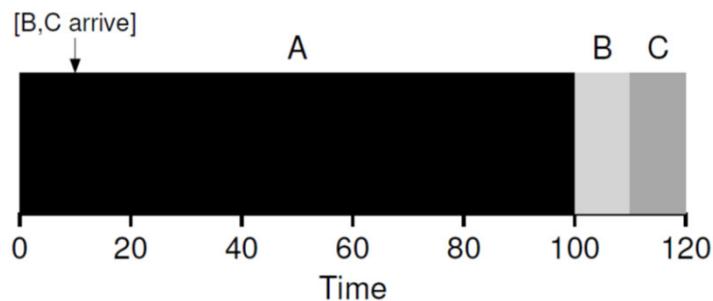
- תחת הנחות שהגדכנו בהתחלה (פחות הנחה 1 עליה כבר ויתרנו) ניתן להוכיח כי SJF הוא האלגוריתם האופטימלי במדד זמן תגובה ממוצע.
 - ההוכחה – בהרצאה.
- האינטואיציה מאחורי SJF היא לתת עדיפות לקלוחות (== תהליכיים) קצרניים כדי שייהו מרווחים (== זמן תגובה נמוך).
 - אנלוגיה ל��פת "עד 10 פריטים" בסופר.
- כעת נסיר את הנחה 2 ("כל התהליכיים מגיעים באותו זמן $t=0$ ").
 - ◀ תהליכיים יכולים להגיע בכל זמן $t > 0$.
- תוכלו לחשב על דוגמה שבת SJF אינם יעילים?

1. כל התהליכים רצים למשך אותו זמן.
2. כל התהליכים מוגשים באותו זמן ($t=0$).
3. אם תחילן התהליך הראשון, אז הוא ירץ עד לסיוםו ללא הפסקות.
4. התהליכים משמשים רק במעבד ולא מביצים I/O'.
5. זמן הריצה של כל התהליכים ידוע מראש.

שוב אפקט השירה...

- תחילן A מגיע בזמן $t=0$ ורץ למשך 100 שניות.
- תהליכי C,B,C מגיעים בזמן $t=10$ וນבקשים לרוץ 10 שניות כל אחד.
- שוב תהליכי קצרים מתעכבים מאחרוי תחילן ארוך.

$$\text{averageResponseTime} = (100 + 100 + 110) / 3 = 103.33$$



1. כל התהיליכים רציש למשך אותו זמן.
2. כל התהיליכים מוגשים באותו-זמן (0=0).
3. אם תחלר התחליל לרוץ, אז הוא ירוץ עד פסותו ללא הפסיקות.
4. התהיליכים משמשים רק בעקבד לא מבצעים O/I.
5. זמן הריצה של כל התהיליכים ידוע מראש.

הפתרון: הפקעה

- כדי לפתור את הבעיה, נסיר את הנחה 3 ("אם תחליך התחליל לרוץ, אז הוא ירוץ עד לסיומו ללא הפסיקות").
◀ מרכיבת הפעלה יכולה להפיקיע את המעבד מהתחליך רץ, וכך:
- לעצור את התהילין הנוכחי ולקורא לתחליך אחר במקומו.
- המעבר בין התהיליכים נקרא "החלפת הקשר".
- תזכורת: המנגנון שמאפשר הפסקה הוא פסיקות שעון.

1. כל התהליכים רצויים למשך אותו זמן.
2. כל התהליכים מוציאים באותו זמן (0=0).
3. אם תהליך התחיל לרוץ, אז הוא ירוץ עד לפניו ללא הפרעה.
4. התהליכים משתמשים רק בזיכרון ולא מביצעים O/I.
5. זמן הריצה של כל התהליכים ידוע מראש.

אלגוריתם SRTF

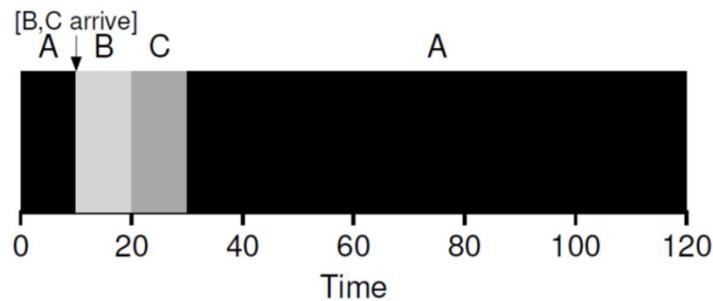
- SRTF = shortest remaining time first
- נקרא גם:
 - אופן פעולה: כמו SJF, אבל עם הפיקעות.
 - בכל פעם שתהליך חדש מגיע למערכת, SRTF מחשב למי בין התהליכים (כולל התהליך החדש) נותר כדי פחות זמן לרוץ, ובוחר את התהליך זהה לריצה.
- תחת ההנחה החדשות, ניתן להוכיח כי SRTF אופטימלי במדד זמן התגובה הממוצע.
- בתוספת הנחה כי זמן החלפת הקשר הוא אפסי.

1. כל התהליכים רצים למשך אותו זמן.
2. כל התהליכים מוגשים באותו זמן ($t=0$).
3. אם תהליך התחיל לרוץ, אז הוא ירוץ עד לפניו ללא הפסוקות.
4. התהליכים משמשים רק במעבד ולא מביצים O/I.
5. זמן הריצה של כל התהליכים ידוע מראש.

דוגמה SRTF

- תהליך A מגיע בזמן $t=0$ ורץ למשך 100 שניות.
- תהליכי B,C מגיעים בזמן $t=10$ וմבקשים לרוץ 10 שניות כל אחד.

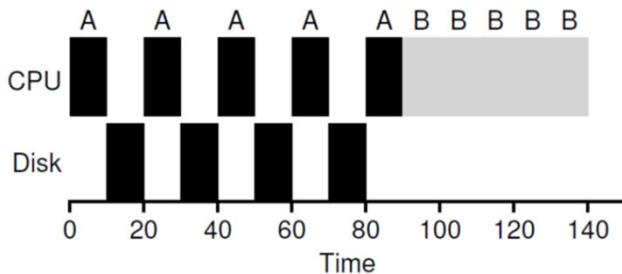
$$\text{averageResponseTime} = (10 + 20 + 120) / 3 = 50$$



1. כל התהליכים רצים למשך אותו זמן.
2. כל התהליכים מוגשים באותו זמן (0=טום).
3. אם תחלר התחיל לרשץ, אז הוא יירץ עד לפיזוטו ללא הפסקות.
4. ההתקלים משאמשים רק במעבד ולא מבצעים O/I.
5. זמן הריצה של כל התהליכים ידוע מראש.

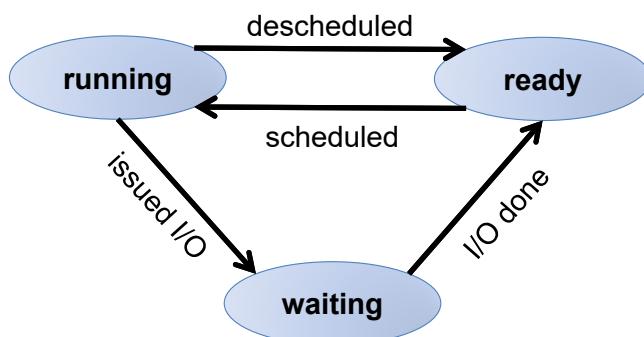
שילוב התקני O/I

- כתע נסיר גם את הנחה 4 ("התהליכים משתמשים רק במעבד ולא מבצעים O/I").
- ← התקנים ניגשים להתקני O/I, לדוגמה דיסק או כרטיס רשת.
- גישה להתקני O/I אורך זמן רב (במונחי מעבד) – מספר מילישניות או יותר.
- תחיל שמתוין ל-O/I לא משתמש במעבד – בעית עילוות.



ניצול טוב יותר של המשאבים

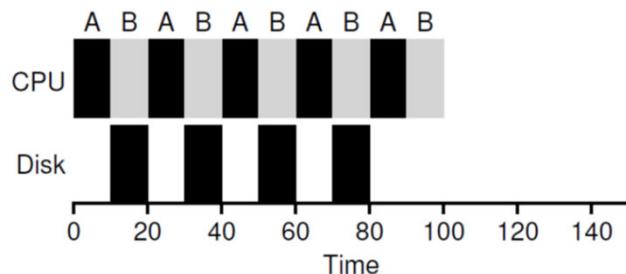
- כasher התהיל'ר הנוכחי מבקש O/I – אלגוריתם היזיון יעביר אותו למצוב המתנה ויזמן תהיל'ר אחר במקומו.
- כasher ההתקן מסיים את פעולתו ושולח פסיקה למעבד – אלגוריתם היזיון יחזיר את התהיל'ר למצוב מוקן לריצה (לא בהכרח יריץ אותו).



1. כל התהליכים רצויים למשרket אותו זמן.
2. כל התהליכים מוציאים באותו זמן (0=0).
3. אם תהליך התחיל ברוץ, אך הוא יפסיק עד לפניו ללא הפסוקות.
4. ההקלילים משתמשים רק בעקבות ולא מבצעים O/I.
5. זמן הריצה של כל התהליכים ידוע מראש.

דוגמה נוספת של SRTF

- תהליך A רץ 50ms בסך הכל.
בכל 10ms הוא מבצע O/I שארך גם 10ms.
- תהליך B גם רץ 50ms בסך הכל, אבל לא מבצע O/I בכלל.
- הגישה המקובלת היא להתייחס לכל ת-משימה של A כאלו תהליך עצמאי באורק 10ms. איך יראה זימון תחת אלגוריתם SRTF?



בדוגמה המופיעה בשקף אלגוריתם SRTF אופטימלי גם מבחינת אינטראקטיביות:
תהליך A (שהוא אינטראקטיבי כי הוא ממතין להתקני O/I כמו מקלדת או עכבר) מקבל עדיפות על פני תהליך B.

נשารנו רק עם הנחה 5...

- "זמן הריצה של כל התהליכים ידוע מראש".
- ההנחה הזאת הגיונית במקרים מסוימים.
- למשל, במקרה של batch scheduling כפי שלמדתם בהרצאה:
- הרבה משתמשים עובדים על מחשב-על (supercomputer) בעל הרבה ליביות.
- המשתמשים שלוחים "עבודות" (jobs) לתור הריצה.
- אלגוריתם הזמן משਬץ את העבודות על הליביות הפנויות.
- המשתמשים נדרשים לספק הערכה בזמן הריצה של העבודות שהם שלוחים.
- עבודה שחרוגת זמן הריצה שהוגדר לה – נעצרת ע"י אלגוריתם הזמן.

האם כדאי למשתמש לספק הערכה גבוהה כדי להבטיח שהעבודה שלו תסתיים בצורה תקינה?

תשובה: המשתמש ישתדל להעיר את זמן הריצה בצורה מדויקת כי הערכה גבוהה או נמוכה מדי עלולה לפגוע בו.

מצד אחד, זמן ריצה גדול יבטיח שהעבודה תסתיים בצורה תקינה.
מצד שני, זמן ריצה קצר יתן לו עדיפות ברוב אלגוריתמי הזמן (למשל SJF,SRTF,SJF) שمعدיפים עבודות קצרות.

(ברוב המקרים תוכנות חישוב מדיעות מתוכנותות לשמר את תוכנות הבניינים במהלך הריצה, כך שאם הריצה נקטעת ניתן להמשיך אותה מקבצי הבניינים כדי לחסוך זמן.)

Batch scheduling

- אלגוריתמי batch scheduling מניחים את הנחות 3,4,5 שראינו:
 - .3 אם תחילת התחליל לרווח, אז הוא יירוץ עד לסיוםו ללא הפסקות.
 - .4 התהליכים משתמשים רק במעבד ולא מבצעים O/I.
 - .5 זמן הריצה של כל התהליכים ידוע מראש.
- שימושו לב: אנחנו הנחנו עבודות סדרתיות, אבל בהרצתה מטפלים גם בעבודות מקביליות (= רצות על מספר ליבות במקביל).
- רוב התוצאות התיאורטיות כבר לא תקפות לעבודות מקביליות.
 - למשל, SJF כבר לא אופטימלי במידע זמן המתנה ממוצע.
- אבל האינטואיציה שקיבלנו בעבודות סדרתיות בדרך כלל נשמרת.

מדד חדש: זמן המתנה

- עד כה למדנו מספר אלגוריתמי זימון והשוינו ביניהם לפי מדד "זמן התגובה":

$$\text{responseTime} = \text{terminationTime} - \text{arrivalTime}$$

- כעת נציג מדד חדש – "זמן המתנה":

$$\text{waitTime} = \text{startTime} - \text{arrivalTime}$$

- כמו קודם, המדד יהיה זמן המתנה הממוצע על פני כל התהיליכים.

- איזה מדד קובע את הביצועים?

- התשובה תלוי בעומס...

נהוג לסוג תהליכיים לשני סוגים

תהליך אינטראקטיבי I/O Bound

- מעוניין בזמן המתנה נמוך.
latency sensitive.
- דוגמה: נגן סרטים שמחליף 60 פריטים בשנייה.
- בדרך כלל מותר על המעבד מרצונו אחריו פרק זמן קצר בגל המתנה לפועלות O/I.

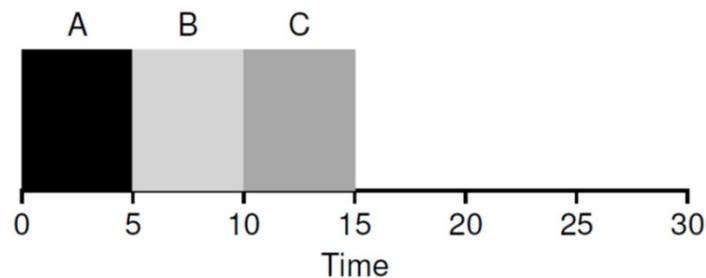
תהליך חישובי CPU Bound

- מעוניין בזמן **תגובה** נמוך.
throughput sensitive.
- דוגמה: סקריפט python שמנתח נתונים ע"י חישובים אלגבריים.
- בדרך כלל לא מותר על המעבד מרצונו אלא מופקע.

לא מצטיינים בזמן המתנה SRTF / SJF

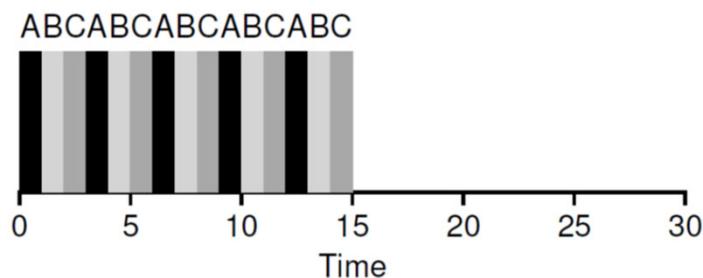
- נניח שלושה תהליכים A,B,C מגיעים באותו הזמן.
- כל תהליך רץ למשך 5 שניות.
- מה יהיה זמן המתנה תחת אלגוריתם SJF או SRTF?

$$\text{averageWaitTime} = (0 + 5 + 10) / 3 = 5$$



אלגוריתם Round Robin

- במקום להריץ תהליכיים עד לסיוםם, האלגוריתם מריץ כל תהליך במשך פיסת זמן מסוימת (time slice או quantum), ואז עובר להריץ תהליך אחר למשך אותה פיסת זמן, וכן הלאה...
 - אם נניח כי הקוונטים הוא 1s, אז: $\text{averageWaitTime} \approx 2\text{s}$
- לצורך חישוב זמן התגובה, נתיחס לכל פיסת זמן כאלו תהליך עצמאי באורך 1s אשר מגיע ברגע שפיסת הזמן הקודם הסתיימה.



שיקולים בבחירה הקוונטום

- הקוונטום חייב להיות כפולה של הזמן בין פסיקות שעון.
 - למשל, אם פסיקות שעון מגיעה כל 10ms ,
 ... $10\text{ms}, 20\text{ms}, 30\text{ms}$...
- למה עדיף קוונטום נמוך?
 - זמן המתנה נמוך \leftarrow יותר אינטראקטיביות.
- למה עדיף קוונטום גבוה?
 - פחות החלפות הקשר \leftarrow ביצועים טובים יותר.
- למשל נניח כי הקוונטום הוא 10ms וזמן החלפת הקשר הוא 1ms .
- מה התקופה של אלגוריתם RR במקורה זהה?

תשובה: 10% מזמן המעבד מתבצע על החלפות הקשר.
תקורה = משאבים הנחוצים כדי לבצע משימה אך אינם תורמים באופן ישיר לביצוע המשימה. במערכות מחשבים, המטרת היא להפחית את התקורה ככל שניתן.
לדוגמא: שמירת רגיסטרים על המחסנית במהלך קריאה לפונקציה היא התקורה על המעבד (וגם קצת על הזיכרון של המחסנית).

פשרה בין זמן תגובה לזמן המתנה

RR

- מנסה להביא למינימום את זמן המתנה הממוצע.
- אבל משיג זמן תגובה ג clue.
- האלגוריתם אינו צריך לדעת מראש את זמן הריצה של כל התהליכים.

SJF, SRTF

- מנסים להביא למינימום את זמן התגובה הממוצע.
- אבל משיגים זמן המתנה ג clue.
- האלגוריתמים האלה מניחים כי זמן הריצה של כל התהליכים ידוע מראש.

אלגוריתמי זמן של מערכות אמיתיות (למשל CFS של לינוקס) ינסו לשלב בין שני הסוגים לעיל.

הפקיד



זמן תהליכיים בלינוקס

שני סוגי תהליכיים בלינוקו

תהליכיים "רגילים" (conventional)

- אינם רגישים בזמן התגובה ויכולים לסתור עיכובים במידה והמערכת עומסה (כלומר אם רצים עוד הרבה תהליכיים במקביל).
- דוגמה: תהליך שמנתח את המידע שנאסף בטיסה לאחר הטיסה.
- דוגמה: תוכנית-hypervisor Powerpoint. בה אתה רואים שקופית זו.

תהליכי זמן-אמת (real-time)

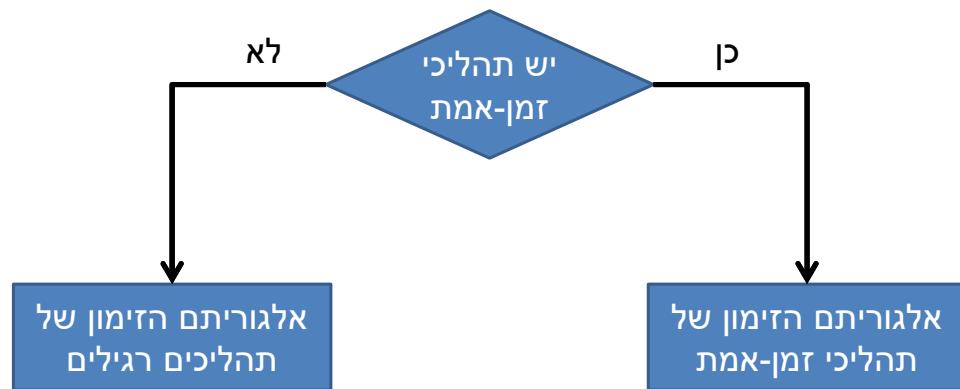
- נדרשים לעמוד באילוצים קשוחים על זמן התגובה, ללא תלות בעומס על המערכת.
- דוגמה: תוכנת הטיס האוטומטי במטוסים.
- רק משתמש-על (root) יכול להגדיר תהליך **זמן-אמת ע"י** שינוי העדיפות שלו.

Example of a real-time process that runs on every Ubuntu system: the “migration” process, which distributes processes across CPU cores (a.k.a. load balancing).

There is one migration process per processor core: migration/1, migration/2, migration/3, ...

שני סוגי תהליכי בלינקו

- לכל אחד מסוגי התהליכים אלגוריתם זימון שונה.
- תהליכי רגילים אינם זוכים לרווח אם יש תהליכי זמן-אמת מוכנים לריצה (כלומר הנמצאים במצב TASK_RUNNING).



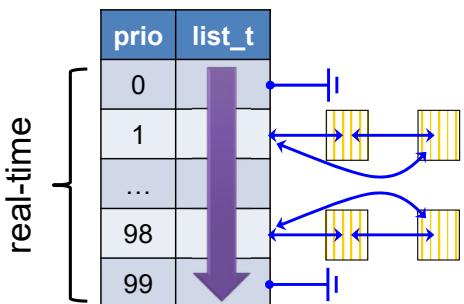
זיכרון: תורי ריצה ותורי המتنה

- התהליכים המוכנים לריצה (מצב **TASK_RUNNING**) נשמרים במבנה נתונים הקרי **runqueue** (טור ריצה).
 - לכל ליבת מעבד יש טור ריצה (מבנה **runqueue**) משלها.
 - בכל רגע נתון, תהליך יכול להימצא בטור ריצה אחד לכל היוטר.
 - אלגוריתם הזמן בחור את התהליך הבא לריצה על המעבד מתוך טור הריצה של אותו מעבד.
- לינוקס מעבירה תהליכי בין טורי ריצה כדי לאזן עומסים אם היא מזזה שיש הרבה תהליכים במעבד אחד לעומת מעבד שני. נושא זה מעבר לחומר הקורס.
- שימוש לב: תהליכי המتنה (== נמצאים בתורי המتنה) אינם נמצאים בתורי הריצה ואין להם קיימים מבחינות אלגוריתם הזמן.

זמן תהליכי זמן אמת בלינוקס

אלגוריתם היזיון של תהליכי זמן-אמת

- כל התהליכים המוכנים לריצה (מצב TASK_RUNNING) נשמרים בתור הריצה: מערך תורים באורך 100, תור לכל עדיפות מספרית.
- כל תהליך נמצא בתור אחד בלבד.
- התהליכים מזומנים לפי **עדיפות** – תהליכי בעדיפות נמוכה יזומנו רק אחרי שהסתיימו כל התהליכים בעדיפות הגבואה.
- מספר גבואה == עדיפות נמוכה.



- האלגוריתם תמיד בוחר לריצה את התהליך שנמצא בראש התור עם העדיפות הגבואה ביותר.

מדיניות זימון של תהלייר

- לכל תהלייר זמן-אמת יש מדיניות זימון (scheduling policy).
SCHED_RR או SCHED_FIFO.
- נקבעת ע"י המשתמש באמצעות קריאות מערכת sched_setscheduler().
- מדיניות הדימון** – תשפייע על כמה זמן ריצה כל תהלייר יקבל ואופן עבודה התור.

מדיניות זימון של תהליכי זמן-אמת

SCHED_RR

Round Robin

- חלוקת זמן בין כל התהליכים המוכנים ליריצה בעלי העדיפות הטובה ביותר.
- כל תהליך מקבל קוונטום (פיסת זמן) בתורו, לפי סדר מעגלי.
- **שימוש לב:** תהליכי מדיניות זימון RR SCHED עלולים "להיתקע" בתור אחרי תהליכי מדיניות זימון SCHED_FIFO.

SCHED_FIFO

First in First Out

- זימון לפי סדר הגעה.
- תהליך FIFO מוגדר על המעבד רק אם:
 - הוא יוצא להמתנה (למשל בغال O/I) – בעtid יחזור לסוף התור.
 - הוא קורא לקריאת המערכת sched_yield – עבר מיד לסוף התור (נשאר בתור הריצה).
- תהליך FIFO מופקע רק ע"י תהליכי זמן-אמת אחר עדיף יותר.

תהליכי בעדיפות זהה לא יכולים להפיקיע את המעבד מהתהליך FIFO, וכך אם הוא לא מוגדר על המעבד עצמו הוא פשוט יירוץ לנצח. לעומת זאת, תהליכי RR מקבלים פיסת זמן כלשהו שבסיוםה מפסיקים מהם את המעבד. לכן אם יש קבוצת תהליכי RR באותה בעדיפות כמו של תהליך FIFO, הם עלולים להיתקע אחרי עד שהוא יותר על המעבד.

תהליך	A	B	C	D
זמן הגעה	1	2	2	3
עדיפות	30	31	30	30

דוגמה

- אם כל התהליכים במדיניות SCHED_FIFO :
 $\underbrace{A\dots}_{q}, \underbrace{C\dots}_{q}, \underbrace{D\dots}_{q}, \underbrace{B\dots}_{q}$
until finished until finished until finished
- אם כל התהליכים במדיניות SCHED_RR :
 $\underbrace{A\dots}_{q}, \underbrace{C\dots}_{q}, \underbrace{D\dots}_{q}, \underbrace{A\dots}_{q}, \underbrace{C\dots}_{q}, \underbrace{D\dots}_{q}$
- לא קשר למדיניות הזמן, תהליך B יירוץ רק כאשר שלושת התהליכים A,C,D יסתינו ו/או לא יהיו מוכנים לריצה.

הוספת תהליכיים לתור הריצה

- תהליכיים חדשים ותהליכיים שחזרו מהמתנה יכנסו לסוף התור המתאים לעדיפותם.
- שאלת: מה היתרונות בלהכניס לסוף התור לעומת תחילת התור?
 - אם המערכת לא עמוסה (כלומר מעט תהליכיים) – זה כמובן לא משנה.
 - תשובה:
 1. הוספת תהליכיים בסוף התור שומרת על הוגנות כי תהליכיים שהגיעו קודם יריצו קודם.
 2. הוספת תהליכיים בתחילת התור הייתה עלולה ליצור הרעה אם תהליכיים חדשים ממשיכים להגיא ותהליכיים בסוף התור לא זוכים ל clue.

חישוב ה-time slice

- כאמור, לכל תהליך במדיניות RR_SCHEDULE מוקצב פרק זמן לשימוש במעבד – time slice או quantum.
- ה-time slice מוגדר ביחידות של פסיקות שעון.
 - בכל פסיקת שעון ערכו קטן ב-1.
 - כאשר הוא מגיע ל-0, התהליך סיים את הזמן שהוקצב לו.
- time slice מוקצה לתהליך במאקרו TASK_TIMESLICE.
 - הчисוב עובר מיחידות של מיל-שניות ליחידות של מספר פסיקות שעון.
 - $Z = \text{מספר פסיקות השעון בשניה.}$

```
#define RR_TIMESLICE      (100 * HZ / 1000)
```

זמן תהליכיים רגילים בLinux

החומר בפרק זה מבוסס על פרק 9.7 (The Linux Completely Fair Scheduler (CFS)) מהספר OSSTEP.

הabolizia של זימון תהליכיים בלינוקס

אלגוריתם הזימון של גרסה 2.4 – נלמד בהרצאה.
פועל בסביבות ליניארית (N)O ולכן **לא סקלריאלי**.

אלגוריתם הזימון של גרסה 2.6 – לא נלמד בקורס.
פועל בסביבות קבועה (1)O, אבל בפועל **איטי מאוד**
בגלל חישובים מורכבים שמנois לסתוג בין תהליכיים
איןטראקטיביים לתהליכיים חישוביים.

אלגוריתם הזימון החל מגרסה 2.6.23 – נלמד
בתרגולים.
פועל בסביבות לוגריתמית (Nlog)O וגם **מהיר פועל**.

Completely Fair Scheduler (CFS)

- אלגוריתם הזמן של גרעין לינוקס החל מגרסת 2.6.23 הוא CFS.
- פותח כדי להציג:
 - יעילות – האלגוריתם מבזבז מעט זמן על קבלת החלטות.
 - מדדיות (scalability) – הביצועים מדרדרים בצורה מתונה יחסית כאשר מספר התהליכים גדול.
- במקום לנסוט להקטין את הזמן המתנה או זמן המתנה, CFS פשוט מנסה להיות הוגן ולתת לכל תהליך נתח שווה של זמן המעבד.
- עד כדי עדיפיות: תהליכי בעדיפות גבוהה יותר יקבלו נתח גדול יותר.
- לצורך פועלתו, אלגוריתם CFS מגדיר מושג חדש:
זמן ריצה וירטואלי.

זמן ריצה וירטואלי (vruntime)

- כאשר תהליך רץ, הוא כובר לעצמו זמן ריצה וירטואלי.
- באופן אידיאלי, זמן הריצה הוירטואלי שווה בין כל התהליכים בכל נקודת זמן.
- אבל באופן מעשי, רק תהליך אחד יכול לרווח על המעבד בכל רגע נתון, ולכן יהיו הפרשים בין התהליכים השונים.
- כאשר CFS מזמן תהליך לריצה הוא יבחר את התהליך בעל זמן הריצה הוירטואלי הנמוך ביותר (כדי להיות הוגן).
- האלגוריתם שומר בכל רגע את המקסימום והמינימום של זמן הריצה הוירטואלי על-פני כל התהליכים המוכנים לריצה – מיד נבין למה.

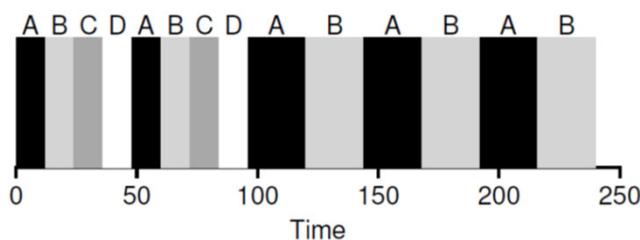
אופן פעולה CFS

- האלגוריתם קובע טווח זמן שבמהלכו הוא ינסה להריץ את כל התהליכים (בדומה ל- epoch של אלגוריתם RR):
$$\text{sched_latency} = 48 \text{ ms}$$
- האלגוריתם מקצה לכל תהליך פיסת זמן (בדומה לקוונטום של אלגוריתם RR) שבו הוא מקבל את המעבד. אם יש N תהליכים במערכת וכולם באוטה עדיפות, הקוונטום של כל תהליך יהיה:
$$Q_i = \text{sched_latency} / N$$
- כאשר תהליך מסיים את הקוונטום שלו, האלגוריתם בוחר לריצה את התהליך בעל זמן הריצה היררכטי הנמוך ביותר בעז.

שימוש לב: הקוונטום המוחשב באלגוריתם CFS אינו בהכרח יוצא כפולה של תדריות פסיקות השעון. זאת לא בעיה כי CFS סופר את זמן הריצה של תהליכי בצורה מדויקת (ריזולוציה עדינה של ננו-שניות) באמצעות השעון הפנימי של המעבד. גם אם תהליכי ירוויח כמו מייל-שניות בטוחה קצר, האלגוריתם יחלק את זמן המעבד בצורה הוגנת בטוחה הארוך.

דוגמה הרצה

- נניח כי במערכת יש ארבעה תהליכיים A,B,C,D.
- از הקוונטום של כל תהליך יהיה 12ms.
- cut נניח כי לאחר שני epochs התהליכיים C,D מסתיימים.
- از התהליכיים הנוגדים B,A ממשיכים לroz עם קוונטום של 24ms.
- הזמן של CFS נראה כמו RR עם קוונטום דינמי:



מה קורה במערכת עמוסה?

- אם מספר התהליכים N במערכת גבוהה, המערכת עלולה לסבול מהחלפות הקשורות ופגיעה בביצועים.

- לכן מוגדר גם זמן מינימום על הקוונטום:

$$Qi \geq \text{min_granularity} = 6 \text{ ms}$$

- לדוגמה, אם יש 10 תהליכים במערכת, אז הקוונטום של כל אחד אמור להיות:

$$Qi = 48 \text{ ms} / 10 = 4.8 \text{ ms}$$

- בפועל, כל תהליך יקבל ms 6 ולכן משך הסיבוב שבו כל התהליכים ירצו יהיה:

$$\text{epoch} = 60 \text{ ms} \geq \text{sched_latency}$$

הבדלים בין RR ו-CFS

RR

- .1. כאשר תהליך מסיים את הקונטום שלו, RR בוחר לሪיצה את התהליך הבא ברשימה המעגלית.
- .2. הקונטום סטטי ואינו תלוי במספר התהליכים במערכת.

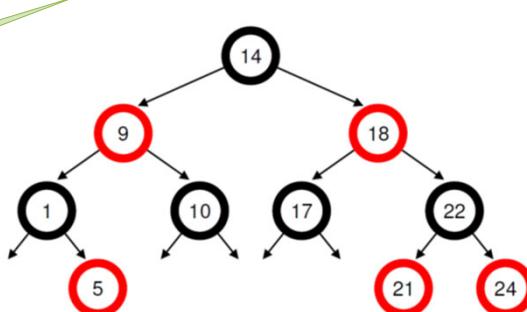
CFS

- .1. כאשר תהליך מסיים את הקונטום שלו, CFS בוחר לሪיצה את התהליך בעל זמן הריצה היורטואלי הנמוך ביותר בעז.
- .2. הקונטום משתנה בהתאם לדינמי בהתאם במספר התהליכים במערכת.

מבנה הנתונים של CFS

- עץ אדום-שחור – עץ חיפוש בינארי מאוזן עם סיבוכיות חיפוש, הכנסה, ומחיקה לוגריתמיות.
- $O(N \log N)$ כאשר N הוא מספר התהליכיים.
- מפתח החיפוש בעץ הוא זמן ריצה הוויירטואלי (vruntime).
- $\text{vruntime} = \max_vruntime$ מאותחלים עם vruntime .

למה?



תשובה: אותה תשובה לשאלת "מדוע מכנים תהליכיים חדשים בסוף התור?"

- כדי לשמר על הוגנות מבחינות סדר הגעה.
- כדי למנוע מתהליכיים חדשים להריעב את התהליכייםקיימים.

יציאה וחזור מהמתנה

- רק תהליכיים מוכנים לריצה נשמרים בעז.
- תהליכיים שבוצעים O/I יוצאים מהעץ וועברים לתור המתנה.
- **תרחיש בעייתי:**
 - שני תהליכיים B,A רצים זה לצד זה.
 - תהליך B יוצא מהמתנה אורך של 10 שניות.
 - כאשר B מתעורר, הוא נכנס לעץ עםvruntime קטן ב-10 שניות מאשר A.
 - לפי CFS, תהליך B יהיה זה שיירוץ ב-10 השניות הבאות.
 - ← הרעבה של תהליך A.
- כדי למנוע את התרחיש הבעייתי הנ"ל, CFS מוסיף תהליכיים שחזרו מהמתנה אורך עםvruntime = min_vruntime .

אלגוריתם CFS מתעדף תהליכי אינטראקטיביים בגלל שהם ממתינים הרבה ושמורים עלvruntime נמוך. במילימ"ש אחריות, תהליכי אינטראקטיביים ישארו בצד שמאל של העץ ולכן הם יזומנו באופן תכופי יותר לריצה ויכמצמו את זמן המתנה שלהם.

עדיפות

- CFS מאפשר למשתמש להגדיר עדיפות לתהילכים וכך לחלק את זמן המעבד בצורה שונה בין התהילכים.
- העדיפות של התהיליך מיוצגת ע"י הערך $+19 \leq \text{nice} \leq -20$.
- ברירת המחדל היא $\text{nice}=0$.
- תהיליך "נחמד" יותר יהיה בעדיפות נמוכה יותר.
- **לכל עדיפות יש משקל:**

```
static const int prio_to_weight[40] = {  
    /* -20 */     88761,      71755,      56483,      46273,      36291,  
    /* -15 */     29154,      23254,      18705,      14949,      11916,  
    /* -10 */     9548,       7620,       6100,       4904,       3906,  
    /* -5 */      3121,       2501,       1991,       1586,       1277,  
    /* 0 */       1024,       820,        655,        526,        423,  
    /* 5 */       335,        272,        215,        172,        137,  
    /* 10 */      110,         87,         70,         56,         45,  
    /* 15 */      36,          29,          23,          18,          15,  
};
```

קצת נסחאות

- נניח שיש במערכת n תהליכים עם עדיפויות: P_1, P_2, \dots, P_n , ומשקלים: W_1, W_2, \dots, W_n .
- נניח כי W_0 הוא המשקל המתאים לעדיפות $0 = nice$.
- אז זמן הריצה הוירטואלי של התהליך i -ו מתקדם לפיה:
$$VR_i = \frac{W_i}{W_0 + W_i} \cdot \Delta T$$
- כאשר ΔT הוא זמן הריצה לפי שעון אמיתי.
- זמן הריצה הוירטואלי זהה לזמן הריצה האמיתית עבור ברירת המחדל $0 = nice$.
- ניתן להוכיח כי הקוונטום של התהליך i -ו הוא:
$$Q_i = \left(\frac{W_i}{\sum W_i} \right) \cdot sched_latency$$

שימוש לב: זמן הריצה הוירטואלי נצבר רק כאשר התהליך רץ (ולא כאשר הוא בתור המתנה /או ממתיין לרוץ בתור הריצה).

דוגמה חישוב

- נניח כי יש שני תהליכיים:
 - $W_1 = 3121$ משקל ← nice = 5 ערך P1
 - $W_2 = 1024$ משקל ← nice = 0 ערך P2
 - נקבל כי תהליך P2 מתקדם (בערך) פי 3 יותר מהר מהתהליך P1:
$$VR_1 += \frac{1}{3} \cdot \Delta T$$
$$VR_2 += \Delta T$$
 - ראינו:
- $$Q_1 = \frac{3}{4} \cdot \text{sched_latency}$$
- $$Q_2 = \frac{1}{4} \cdot \text{sched_latency}$$

תרגול 6

-
- מהי החלפת הקשר (context switch)?
 - מימוש החלפת הקשר בلينוקס
 - יצירת תהליך חדש בلينוקס
 - סיום תהליך בلينוקס

Resources:

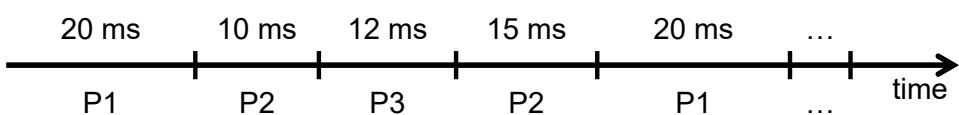
https://www.maizure.org/projects/evolution_x86_context_switch_linux/

TL;DR

- מערכות הפעלה מ Ritchie מספר תהליכי "בו-זמןית" כדי (1) להגדיל את נצילות המעבד ו-(2) להקטין את זמן התגובה שחווה המשתמש.

- האשליה זו מושגת באמצעות **החלפות הקשר** (context switch).

- הגרעין מחליף במתירויות בין ההליכים: מרץ תהליך אחד לפרק זמן קצר (כמה מילישניות) ואז משנה את ביצועו ועובר להרץ תהליך אחר, וכן הלאה.



- היום נלמד איך קורא הקסם של החלפות הקשר בلينוקס!

חזרה קצרה על אט"מ

ארQUITקטורת 46x, קונבנציית קריאה לפונקציות,
קריאות מערכות, פסיקות

רегистרים לשימוש כלל בארכיטקטורת x64

General-Purpose Registers (GPRs)	
RAX	
RBX	
RCX	
RDX	
RBP	
RSI	
RDI	
RSP	
R8	
R9	
R10	
R11	
R12	
R13	
R14	
R15	

63 0

Taken from: <https://www.amd.com/system/files/TechDocs/24592.pdf>

In white: legacy x86 registers, supported in all modes.

In gray: register extensions, supported in 64-bit mode.

רגיסטרים "מיוחדים"

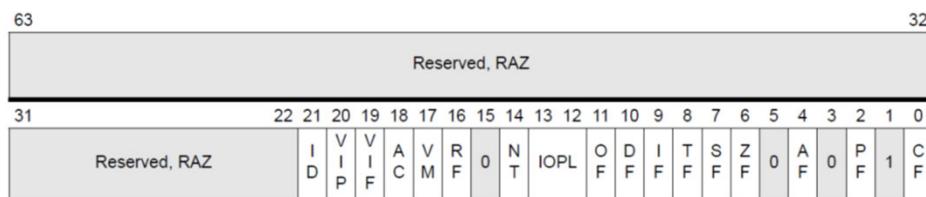
- **RIP** – מצביע על הפקודה הבאה לביוצע.

- כמו כל הרגיסטרים, גם הוא ברוחב 64 ביט.

- **RFLAGS** – שומר את המצב הנוכחי של המעבד, לדוגמה:

- ביט מס' 7 (sign flag) מציין האם תוצאה החישוב האחרון הייתה שלילית.

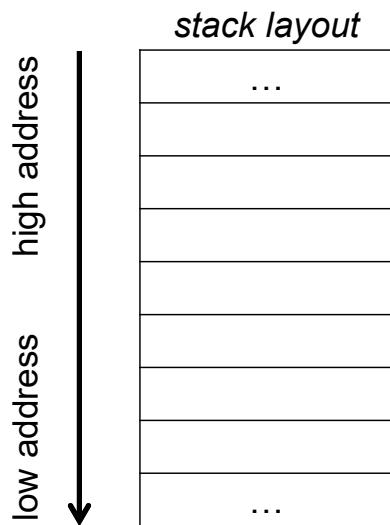
- ביט מס' 9 (interrupt flag) מציין האם המעבד מקבל פסיקות.



- **CS** – שומר את רמת הרשאה הנוכחיית (CPL).

Taken from AMD64 Architecture Programmer's Manual Volume 2: System Programming

מחסנית הקריאה



- כל תכנית מתחזקת **מחסנית**

קריאה במליך ריצתה.

- למשל, כדי לשמור משתנים מקומיים של הפונקציה.

- בעת קראיה לפונקציה התוכנית שומרת על המחסנית:

- את כתובות החזרה מהפונקציה,

- ואת הפרמטרים המועברים לפונקציה.

- אנחנו נציג את המחסנית גדלה למטה (כתובות נמוכות למטה).

קונבנציית לינוקס לקריאה לפונקציות

חוקים של הנקראות
(callee rules)

• בכניסה לפונקציה:

- פתיחת מסגרת חדשה (שמירת `ebp` הישן והצבעה לראש המחסנית).
- הקצאת משתנים מקומיים.
- שבירת הרגיסטרים באחריות הנקראות (`z15—z12, ax, bx`) על המחסנית.

• ביציאה מהפונקציה:

- העברת ערך החזרה ל-`ax`.
- שילפת הרגיסטרים שנשמרו.
- שחרור המשתנים המקומיים.
- חזרה למסגרת הישנה (שליפת `ebp` שנשמר קודם).
- פקודת מכונה `ret`.

חוקים של הקוראות
(caller rules)

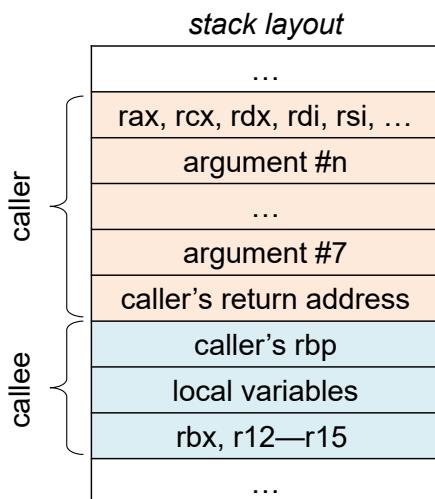
• לפני הקריאה לפונקציה:

- שמירת הרגיסטרים באחריות הקוראות (`z11—z8`) על המחסנית.
- העברת 6 הפרמטרים הראשונים ברגיסטרים (משמאלי ימין): `r9, rdi, rsi, rcx, rdx, r8`.
- העברת שאר הפרמטרים (מעבר לששת הראשונים) על המחסנית.
- פקודת מכונה `call`.

• בחזרה מהפונקציה:

- שילפת הפרמטרים שנחקרו על המחסנית.
- שילפת הרגיסטרים שנשמרו.

קריאה וחזרה מפונקציה



- פקודת המכונה `call` :

- דוחفت את כתובות החזרה (ערכו הנוכחי של `r10`) למחסנית,
- וקופצת לתחילת הפונקציה על-ידי הצבת כתובות הפונקציה ל-`r10`.

- פקודת המכונה `ret` :

- שולפת את כתובות החזרה מראש המחסנית,
- וקופצת לכטבות זו על-ידי הצבתה ל-`r10`.

kernel stack
ss
rsp
rflags
cs
rip
orig_rax
rdi
rsi
rdx
rcx
rax
r8
r9
...
r15

מחסנית הגרעין בזמן טיפול בקריאה מערכת או פסיקה

- בתחילת הטיפול כל הרגיסטרים נשמרים על מחסנית הגרעין באמצעות המacro PUSH_REGS.
- כى הגרעין צריך לשחזר את מצב המעבד לפני קריאת המערכת או הפסיקה.
- בסוף הטיפול כל הרגיסטרים נשלפים באמצעות המacro POP_REGS.
- בין הרגיסטרים נשלף גם ax וכך שגרת הטיפול למעשה מוחזירה את הערך במקרה של קריאת מערכת.

Some registers are missing in the figure: r10, r11, rbx, rbp, r12, r13, r14 . The real name of “PUSH_REGS” is “PUSH_AND_CLEAR_REGS”.

מהי החלפת הקשר?

מהי החלפת הקשר?

- לכל תהליך יש **"הקשר ביצוע"** (execution context) המכיל את כל המידע הדרוש לביצוע התהליך.
 - מחסניות, רגיסטרים, תכולת זיכרון, קבצים פתוחים, ...
- "החלפת הקשר" =
 1. עיצירת הביצוע של התהליך הנוכחי ושמירת הרקשור שלו.
 2. טיענת הרקשור של התהליך הבא לביצוע.
- הקשר התהליך הנוכחי מתחלף – מכאן שם הפעולה "החלפת הקשר".

יתרונות וחסרונות של החלפות הקשר

• ניצול טוב יותר של משאבי המערכת.

- לדוגמה: כאשר תחילה A ממתין לנ נתונים מהדיסק, מערכת החלפה תגרום לו לפנות את המעבד ותקרא לתהילך אחר B במקומו.

• הקטנת זמן התגובה של תהליכי אינטראקטיביים.

- לדוגמה: תחילה A רץ על המעבד, תחילה B ממתין לתווים מהמקלדת. ברגע שתגיע פסיקת מקלדת, היא תטופל בהקשר של תחילה A, ותעיר את תחילה B.
- מערכת הפעלה תפרק את המעבד מתחילה A וזמן לריצה את תחילה B.

• פגיעה בנצחונות המעבד (CPU utilization).

- מערכת הפעלה מב齊זת זמן על שמירת הקשר הנוכחי וטעינת הקשר החדש.

From OSSTEP:

“Note that the cost of context switching does not arise solely from the OS actions of saving and restoring a few registers. When programs run, they build up a great deal of state in CPU caches, TLBs, branch predictors, and other on-chip hardware. Switching to another job causes this state to be flushed and new state relevant to the currently-running job to be brought in, which may exact a noticeable performance cost.”

שני סוגי של החלפת הקשר

החלפת הקשר כפואה (== הפקעה)

- **הגרעין מפרקיע (כלומר, לוקח בכוח) את המעבד מהתהליך,** למשל בעקבות:
 1. פסיקת שעון (מטופלת בשגרה scheduler_tick) אשר מגלה כי הזמן שהוקצב לתהליך הנוכחי אזל.
 2. אירוע אסינכרוני אשר מעיר התהליך בעל עדיפות טוביה יותר מהתהליך הרץ כרגע.
- **לדוגמה:** פסיקת דיסק או שחרור מנעול שתהליך המתין לו.

החלפת הקשר יזומה

- **התהיליך מוותר מרצונו על המעבד**, למשל באמצעות:
 1. קריית מערכת חוסמת (כמו ... , read(), wait()) אשר מוציאאת את התהיליך להמתנה.
 2. קריית מערכת exit() אשר מסיים את התהיליך.
 3. קריית מערכת sched_yield() – קריית מערכת ייעודית לווייתור על המעבד.

החלפת הקשר יזומה נגרמת גם בעקבות פעולות יזומות של התהיליך שגורמות לשגיאה וסיום התהיליך, למשל:
גישה לכתובת לא חוקית בזיכרון (כמו `NULL`), חלוקה באפס, ביצוע פקודת מכונה לא חוקית, וכו'.

פעולות כאלה אינןمسؤولות כהפקעה מפני שהטהיליך היה יכול להימנע מהן, בניגוד למשל לפסקיות שעון שגורמות להחלפת הקשר כפואה.

הפקעה (preemption)

- **בעיה:** תהיליך משתמש עלול לרוץ לנצח (למשל, לולאה אינסופית) ולמנוע את המעבד משאר התהליכים.
- פגיעה בהוגנות (**fairness**) ותגובהיות (**responsiveness**).
- **פתרון:** לינוקס מפרקעה (preempt) את המעבד מהתהליך אחד לטובת תהיליך אחר, בעזרת התקן חומרה מיוחד – **השען**.
- מערכת ההפעלה מבקשת מהשען לשלוח פסיקה במרוחך זמן קבועים כדי להעביר את השיטה למערכת ההפעלה.
 - כל הפסיקות, בפרט פסיקת שעון, **מטופלות במצב גרעין**, ואז הגרען מחליף הקשר אם יש צורך.

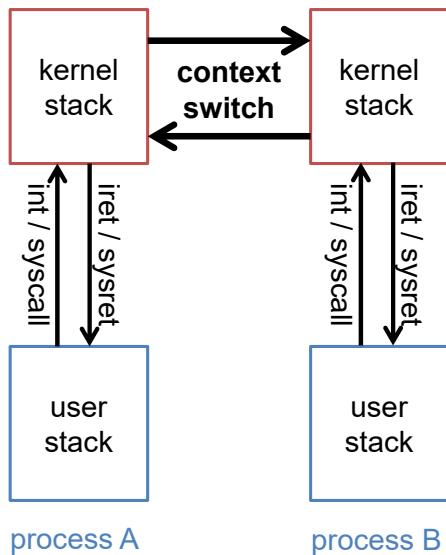
הפקעה במצב גרעין

- הפקעה של תהיליך שרש במצב גרעין היא מסובכת יותר בגלל בעיות סyncron (נלמד בתרגולים בנושא סyncron).
- בגרסאות ישנות של Linux, הגרעין לא היהאפשר להפיקיע את המעבד מהתהיליך שנמצא במצב גרעין.
- אבל החל מגרסה 2.6, גרעין Linux מסוגל להפיקיע את המעבד גם מהתהיליך שנמצא במצב גרעין.
- לדוגמה, הגרעין יכול להפיקיע את המעבד מהתהיליך שנמצא באמצע הביצוע של קריית המערכת (`fork` במצב גרעין).

לקריאה נוספת:

https://en.wikipedia.org/wiki/Kernel_preemption
<https://kernelnewbies.org/FAQ/Preemption>

החלפת הקשר מתרחשת במצב גרעין



- החלפת הקשר דורשת את התערבות מערכת הפעלה, ולכן היא מתרחשת במצב גרעין.
- כדי לשמורת הטרשים, המעבר ממצב משתמש למצב גרעין מתרחש רק בעקבות קריית מערכת או פסיקה (חומרה/תוכנה).
- לב העניין הוא החלפה בין מחסניות הגרעין של שני התהליכים המעורבים.

איך הגריעין מפעיל החלפת הקשר?

כפי שנלמד בתרגול
על זמם התהליכים
(scheduler)

- ע"י קריאה לפונקציה `schedule()` אשר:
 1. בוחרת מי יהיה התהיליך הבא לרכיבה.
 2. מבצעת את החלפת הקשר ע"י קריאה לפונקציה `context_switch()`.
- `schedule()` היא שער הכניסה היחיד להחלפת הקשר בלבד.

- הפונקציה `schedule()` רצה במצב של פסיקות מנטרלות על-מנת למנוע גישה לא מתואמת לתור הריצה (`runqueue`).
- מנטרלים קבלה של פסיקות מעבד לפני הקריאה ל-`schedule()` ומאפשריםשוב את קבלתן לאחר סיום הפונקציה.
 - זה מגן סיכון שנועד להגן על מבני הנתונים הנגישים לפונקציה `schedule()` – פרטיים נוספים בתרגול בנושא סיכון.

דוגמת קוד מטור הגרעין

- המאקרו () wait_event_interruptible מכניס את התהילך הנוכחי להמתנה בתור עד אשר התנאי condition מתקיים.

```
#define wait_event_interruptible(wq_head, condition)
{
    struct wait_queue_entry wq_entry;
    wq_entry->task = current;
    current->state = TASK_INTERRUPTIBLE;
    for (;;) { // infinite loop
        if (condition) break;
        add_wait_queue(&wq_head, &wq_entry);
        schedule();
    }
    remove_wait_queue(&wq_head, &wq_entry);
}
```

איבר חדש
שנוייף לתור

מדוע מותר להעביר
מצבע למשתנה מקומי?

תשובה: כי מוצאים את המשתנה `wq_entry` מטור ההמתנה בסוף הפונקציה, כלומר לפני שהוא נמחק.

המשתנה `wait` אומנם ח' רק בתוך המאקרו, אבל כאשר קוראים לו- (`schedule()`)
התהילך מוקפף והזיכרון שלו נותר ללא שינוי ולכן ניתן עדין לגשת לאיבר `wq_entry`
מהקשרי ביצוע אחרים.

שימוש לב: בשקף מופיעעה גרסת קוד מפשטה כי הקוד האמיתי מסורבל למדי.

הפרק

ASSEMBLY



SYSTEM CALLS



PROCESSES



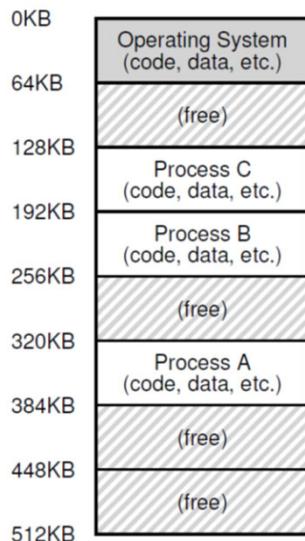
CONTEXT SWITCH



imgflip.com

שימוש החלפת הקשר בلينוקס

היכן נשמר הזיכרון של התהליכים?



- אזור הזיכרון של כל התהליכים (וגם של מערכת הפעלה) חיימ זה לצד זה.
- החלפת הקשר אינה דורשת "שמירה" או "טיענה" של אזור זיכרון, אלא רק החלפת מרחבי הזיכרון.
- במעבדי 64x, מערכת הפעלה מחליפה את מרחב הזיכרון ע"י טיענת ערך חדש לרגיסטר CR3.

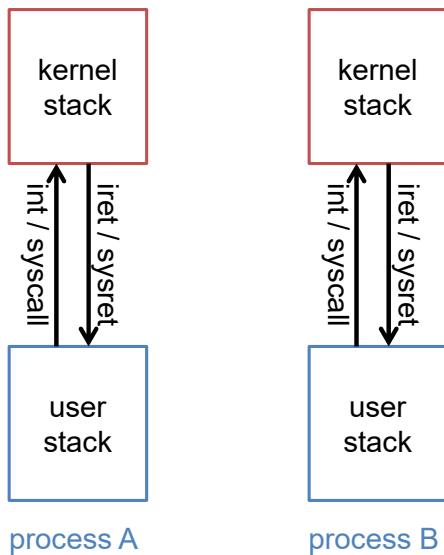
Figure 13.2 from the OSTEP book.

לכארה יש כאן בעית אבטחה כי כל התהליכים חולקים ביניהם את הזיכרון. בפועל, מנגנון הזיכרון הוירטואלי מספק בידוד והגנה בין התהליכים.

היכן נשמר הקשר התהיליך?

- .1. המחסנית, הערימה, הקוד נמצאים בזיכרון.
 - כאמור, אין צורך "לשמור" ו"לטען" אותם מחדש בכל החלפת הקשר.
- .2. נתונים אחרים, למשל הקבצים הפתוחים (file descriptors), נשמרים במתאר התהיליך (the PCB) אשר גם נמצא בזיכרון.
- .3. את הרגיסטרים והדגלים (מצב המעבד) יש לשמר ולטען מחדש לאחר מכון. השמירה והטיענה מתבצעת **במחסנית הגרעין** ובשדה PCB ב-thread.
- .4. בנוסף, יש לשמר פריט מידע נוסף שעד עכשו התעלמו ממנו – את **בסיס מחסנית הגרעין** של התהיליך הנוכחי.

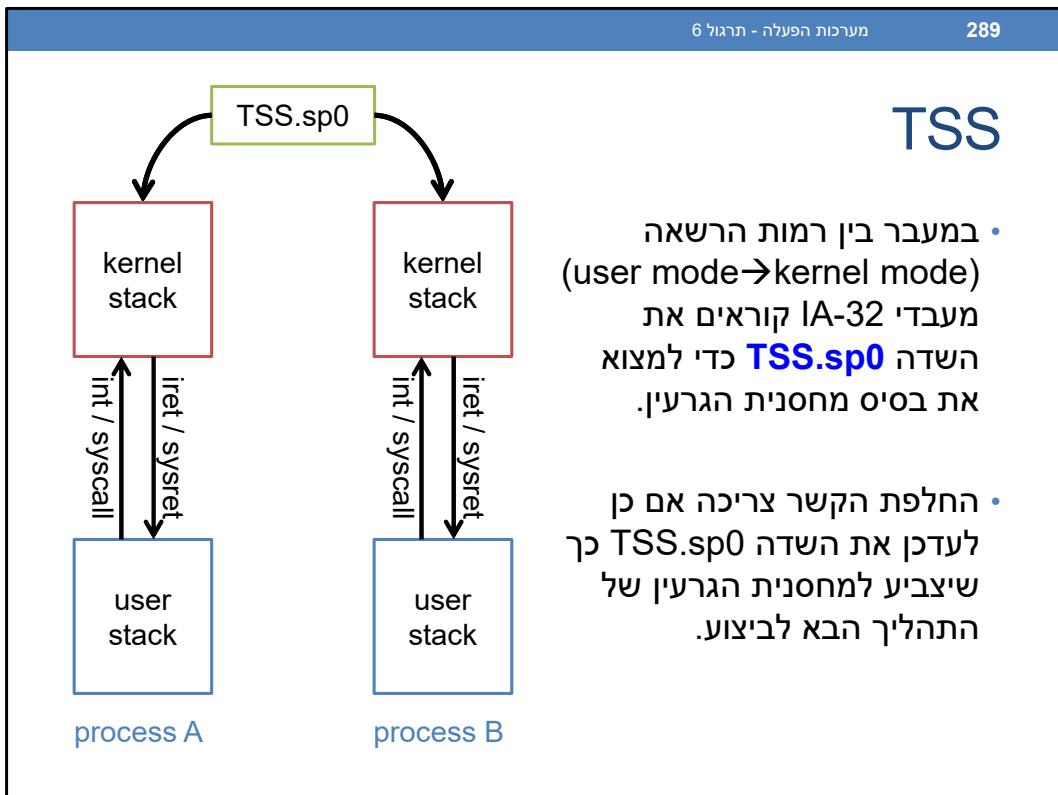
תזכורת

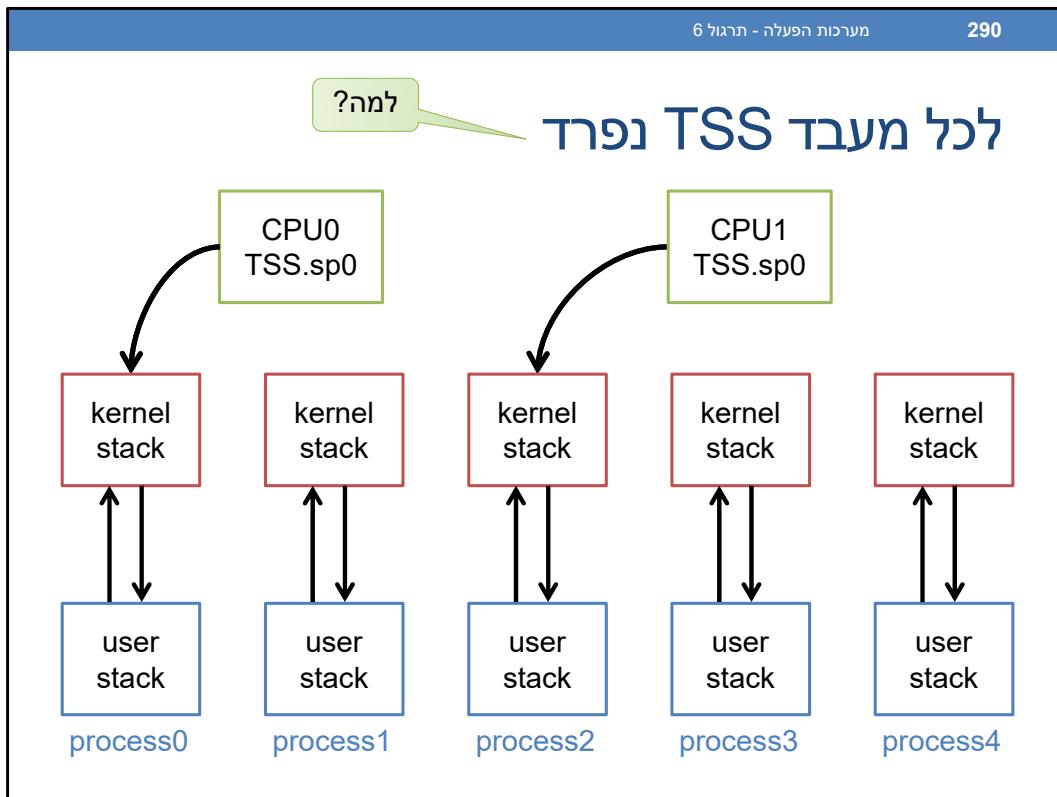


- כasher תהלייר רץ במצב משתמש וatz מתקבלת פסיקה, המעבד מחליף מחסניות ושומר את ההקשר של המשתמש על מחסנית הגרעין.
- לכל תהלייר יש מחסנית גרעין משלהו.**
- שאלה:** איך המעבד יודע איפה נמצאת מחסנית הגרעין של התהלייר הנוכחי?
- תשובה:** באמצעות מבנה מיוחד הנקרא **TSS**.

TSS

- במעבר בין רמות הרשאה (user mode → kernel mode) מעבדי IA-32 קוראים את השדה **TSS.sp0** כדי למצוא את בסיס מחסנית הגרעין.
- החלפת הקשר נדרש אם כן לעדכן את השדה 0 TSS.sp0 כך שיצביע למחסנית הגרעין של התהיליך הבא לביצוע.





Why do need a separate TSS struct for each CPU?

The TSS.sp0 serves as a “register” that points to the base of the current kernel stack. Since all registers are per-CPU, so is the TSS struct.

TSS (Task State Segment)

- TSS (Task State Segment) הוא מבנה נתונים הנמצא בזיכרון הגרעין, ומכיל מידע על הקשר התהיליך הנוכחי המתבצע במעבד.

```
struct tss_struct {  
    ...  
    unsigned long sp0;  
    ...  
};
```

מצביע לבסיס מחסנית הגרעין
של התהיליך הנוכחי במעבד

- TSS מוצבע ע"י רגיסטר מיוחד במעבד הנקרא **TR**.
- במערכת מרובת ליבות מוקצת לכל ליבה מבנה TSS משלها.

מעבד 64x משתמשים במבנה TSS לצורך פעולות נוספות, למשל (לא בחומר הקורס) בקרת גישה לפורטים בפעולות O/I.

שדה thread במתאר התהלייר

- השדה `thread` הוא מבנה מטיפוס `thread_struct` אשר נמצא בתוך .PCB.
- השדה משמש לשימרת חלק מהקשר התהלייר.
- פריט המידע החשוב מבחרינו הוא המצביע לראש מחסנית הגרעין שנשמר ונטען בזמן החלפת הקשר:

```
struct thread_struct {  
    ...  
    unsigned long rsp;  
    ...  
};
```

מוגדר בקובץ הגרעין `. include/asm/processor.h`

שלבי החלפת ההקשר

כללית, לא תלויות ארכיטקטורתית – **context_switch()**

לארכיטקטורה, כתובה באסמבלי צי – **switch_to_asm()**

לארכיטקטורה צי ניגשים למבנה TSS – **switch_to()**

הפונקציה context_switch()

מצביע למתאר התהילך הנוכחי, שמOOTר על המעבד

```
context_switch(..., struct task_struct *prev,  
              struct task_struct *next, ...){  
    ...  
    ...  
    switch_mm(oldmm, mm, ...);  
    ...  
    __switch_to_asm(prev, next);  
    ...  
}
```

מצביע למתאר התהילך הבא, שמקבל את המעבד

החלפת מרחבי הזיכרון

המאקרו ששמור את ההקשר
ה הנוכחי וטוען את ההקשר החדש

Code from: kernel/sched/core.c

<https://elixir.bootlin.com/linux/v4.15.18/source/kernel/sched/core.c#L2757>

(1) __switch_to_asm

```
__switch_to_asm:
    pushq %rbp
    pushq %rbx
    pushq %r12
    pushq %r13
    pushq %r14
    pushq %r15
```

הfonקציה `__switch_to_asm` עומדת לבצע
החלפת הקשר לתהילן חדש ולכך כל
הרגיסטרים במעבד יכולים להשתנות.
אבל הקומpileר לא יודיע את זה!
לכן הfonקציה `__switch_to_asm` צריכה
לשמר בעכמתה את הרגיסטרים שהם
באחוריותה (בתווך הfonקציה הנקרואת) ולשזר
אותם לפני החזרה מהfonקציה.

```
    movq %rsp, prev->thread.rsp
    movq next->thread.rsp, %rsp
```

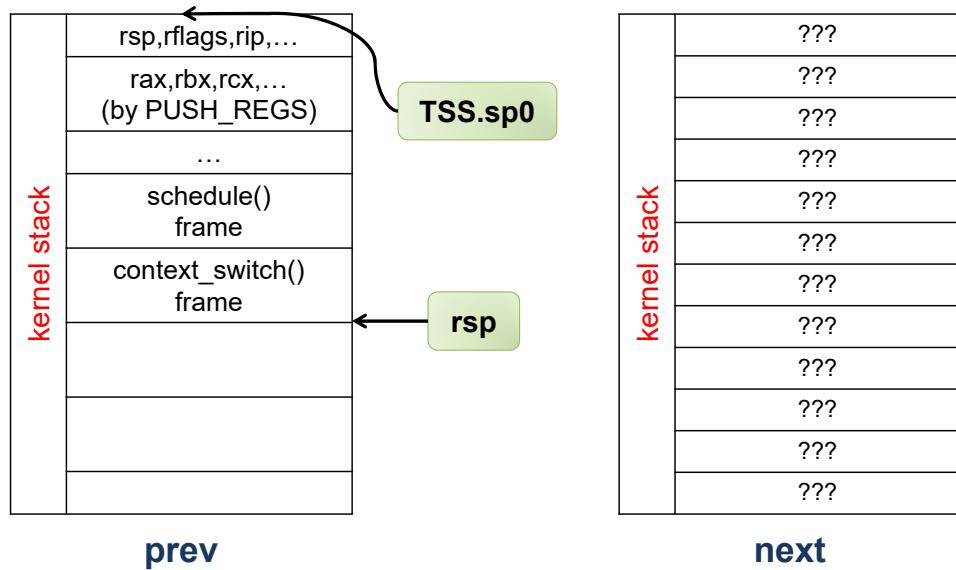
החלפת המוחסניות

בקוד אסמלטי לא ניתן לרשום משתנים ב-C אלא צריך להחליף את
הפרמטרים `next`, `prev`, `prev->thread.rsp` ו-`next->thread.rsp` בההתאמה.

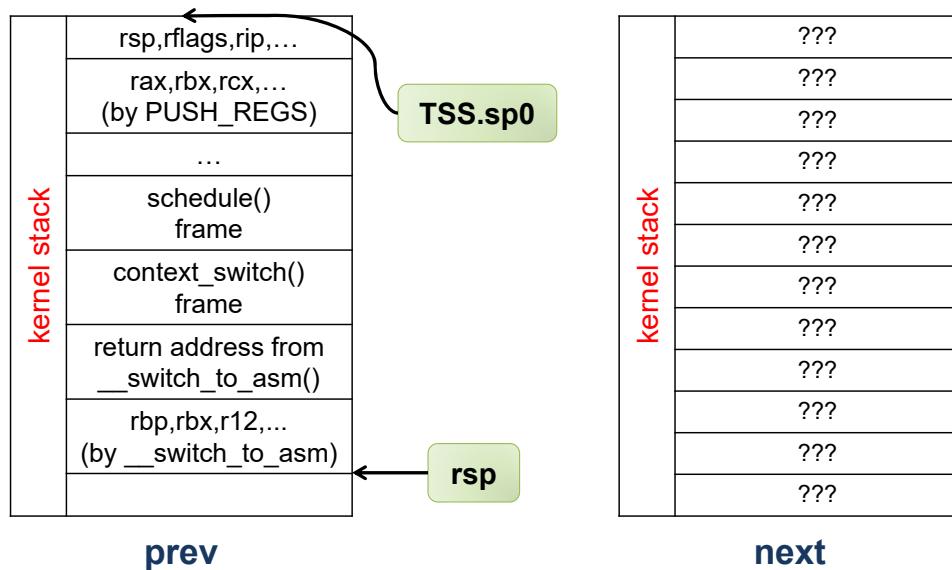
Code from:

arch/x86/include/asm/switch_to.h
arch/x86/entry/entry_64.S

תמונה מצב לפני החלפת ההקשר



תמונה מצב לפני החלפת המחסניות



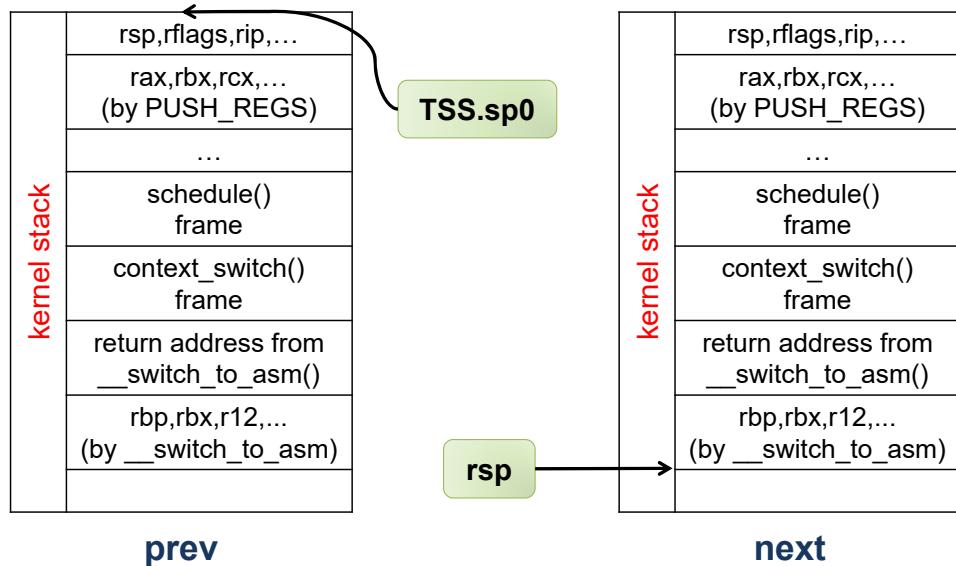
הפונקציה `(2) __switch_to_asm`

- השורה הבאה במקורה מחליפה את **מחסניות הגרעין**, מזו של `prev` לזו של `next` – זו למעשה נקודת החלפת הקשר:


```
movq next->thread.rsp, %rsp
```
- **שאלה:** לאן מצביע `next->thread.rsp` ?
- **תשובה:** תלוי... נפריד לשני מקרים:
 1. **מקרה אתחול:** התהיליך `next` הוא **תהליך חדש שנוצר ע"י** (`fork()`) ועדין לא רץ אף פעם. נחצוץ לדון במצב זה בסוף התרגול.
 2. **מקרה סטנדרטי:** התהיליך `next` כבר רץ **בעבר ועבר החלפת הקשר**. במקרה זה המחסנית של `next` נראה כמו המחסנית של `prev`, כי `prev` הוא **תהליך שעובר עכשו חלפת הקשר!**

אנחנו למשה נוכח באינדוקציה שהקוד מבצע החלפת הקשר תקינה מ-`prev` ל-`next`. האינדוקציה היא על מספר הפעמים N שהטהיליך `next` רץ **בעבר**. עד הבסיס ($N=0$) הוא מקרה האתחול, שבו נראה כי החלפת הקשר פועלת כשרה כאשר התהיליך `next` עוד לא רץ אף פעם. בצעד האינדוקציה נניח `next` כבר רץ **בעבר K פעמים** ואז נראה שnitן לעבור אליו שוב כדי שירוץ $+1$ פעמים.

תמונה מצב במקהה הסטנדרטי



This is for the **standard** case.

(3) __switch_to_asm

```
popq %r15  
popq %r14  
popq %r13  
popq %r12  
popq %rbx  
popq %rbp
```

שחזור הרגיסטרים ממחסנית הגרעין של `next`.
אלו הרגיסטרים ש-`next` שמר כאשר הוא קרא
להחלפת הקשר בעבר.

```
jmp __switch_to
```

קפיצה (`jmp`) במקום קריאה (`call`) לפונקציה. למה?
כתובת החזירה מהפונקציה (`__switch_to()`) כבר
שמורה על המחסנית של `next`.

הfonקציה `switch_to()`

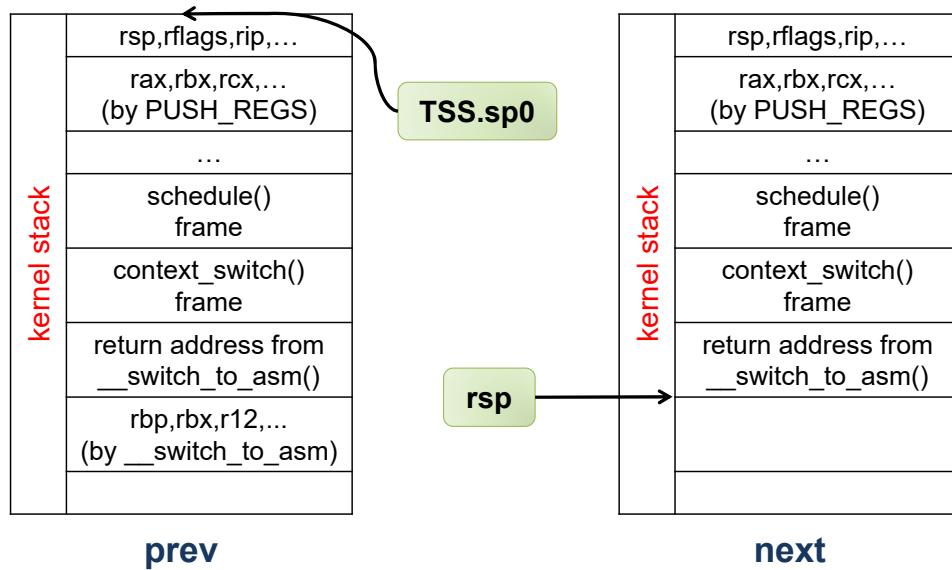
- fonקציה זו משלימה את רצף הפעולות במסגרת החלפת ההקשר.

```
_switch_to(struct task_struct *prev_p,  
           struct task_struct *next_p) {  
    ...  
    update_sp0(next_p);  
    ...  
    return;  
}
```

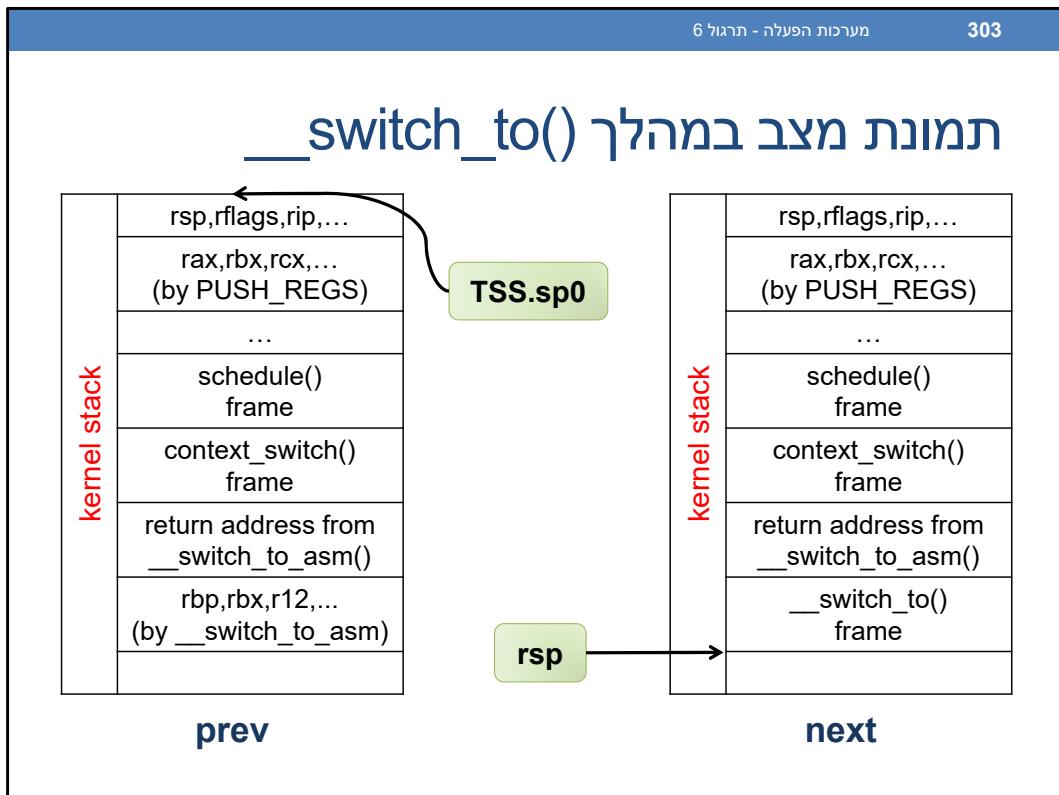
עדכן מצביע מהחונית הגרעין
של התהילך הנוכחי ב-TSS

שליפת כתובות החזרה מראש המחונית

תמונה מצב לפני `__switch_to()`

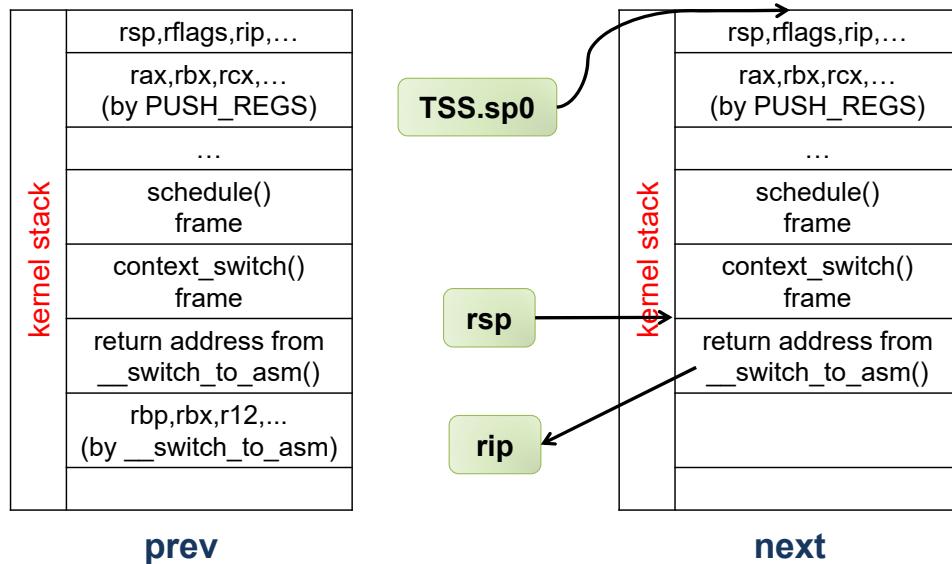


This is for the **standard** case.



This is for the **standard** case.

__switch_to()



This is for the **standard** case.

הfonקציה (2) `__switch_to()`

- **שאלה:** מדוע ה Fonקציה (`__switch_to()`) מופעלת באמצעות קפיצה (`jmp __switch_to`) ולא באמצעות קריאה רגילה (`call __switch_to`) ?
- **תשובה:** פקודת `call` דוחפת למחסנית את כתובות החזרה של השורה הבאה לביצוע, ואנו חנו רצים כתובות חזרה שונה בהתאם ל McKie :
 - במקרה **הẤחול**: הכתובת היא `ret_from_fork` כפי שנראה בהמשך .
 - במקרה **הסתנדרט**: הכתובת היא כתובות החזרה מ- (`__switch_to_asm`) .
 - בשני המקרים, מכיוון ש- (`__switch_to()`) מוגדרת כ Fonקציה לכל דבר, הקוד שלו מסתאים בפקודת `ret` ששולפת כתובות חזרה מהמחסנית (של התהיליך הבא לביצוע) ו קופצת אליה .

יצירת תהילך חדש בLINUKO

או: איך מוממשת קריית המערכת `fork`?

דיון בכיתה – איך היitem עושים את `fork` – מה עליה לעשות?



הfonקציה () do_fork()

- קריית המערכת () fork משתמש בפונקציה פנימית של הגרעין הקוריה () do_fork לבניית ההקשר של התהיליך החדש.
- גם קריות מערכת אחרות לייצור תהילכים (לדוגמה () clone) קוראות ל- () do_fork.
- הפונקציה () do מוצעת את השלבים הבאים:
 - .1 מקצת PCB חדש ומחסנית גרעין חדשה עברו לתהיליך הבן.
 - .2 קוראת לפונקציה () copy_thread, אשר ממלאת את מחסנית הגרעין של תהיליך הבן כך שיראה כאילו הוא קרא לפונקיות המערכת () fork ואז לפונקציה () __switch_to_asm.

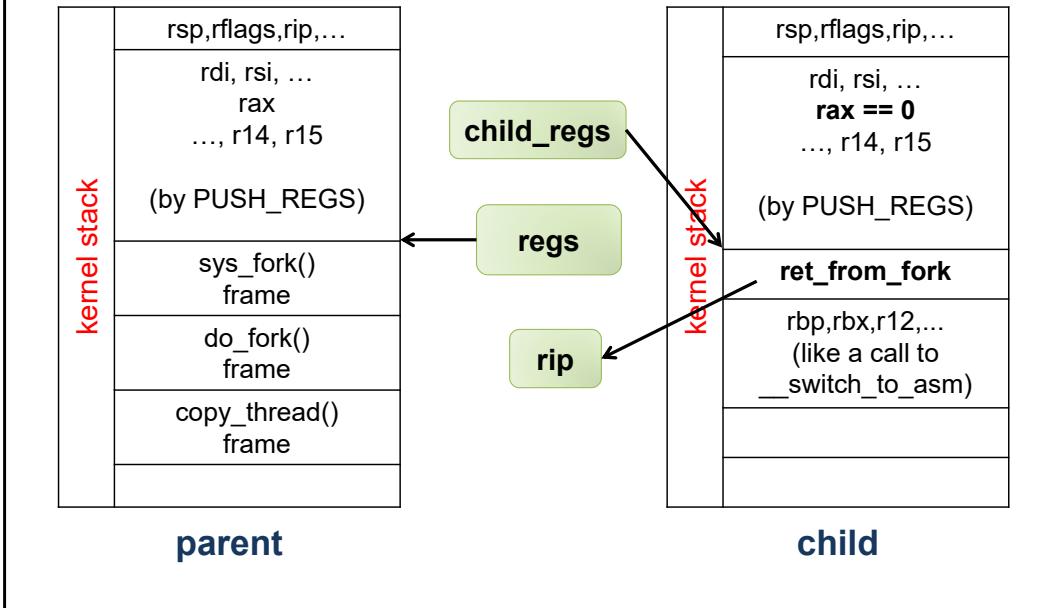
ממומשת בקובץ גרעין kernel/fork.c



הfonקציה () do_fork

- .3. מעתיקת לתהlixir הבן את מרבית הנתונים מ-PCB האב.
- למשל את טבלת הקבצים הפתוחים (file descriptors) ואת שגרות הטיפול בסיגנלים.
- העתקת תכולת הזיכרון מתבצעת בשיטת COW.
- .4. מקשרת את תהlixir הבן ל"בני משפחתו".
- .5. מוסיף את תהlixir הבן לרשימת התהלייכים הגלובאלית וגם לטבלה הערבול PCB→PID.
- .6. מעבירה את תהlixir הבן למצב TASK_RUNNING ומכניסה אותו ל-`newrunq`.
- .7. לסיום, הfonקציה מחזירה את ה-pid של תהlixir הבן, ועריך זה מוחזר גם מטהlixir האב.

הfonקציה(`copy_thread()`



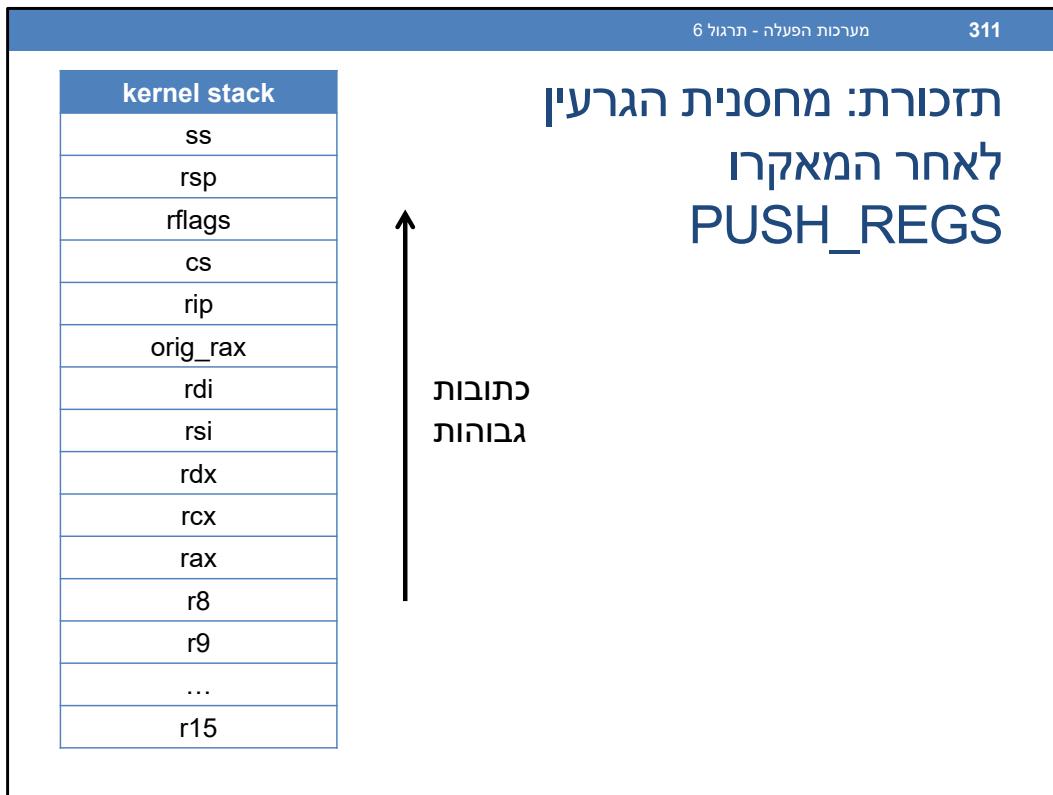
תזכורת: הפונקציה `__switch_to_asm`

```
popq %r15  
popq %r14  
popq %r13  
popq %r12  
popq %rbx  
popq %rbp
```

שחזור הרגיסטרים מחסנית הגרעין של `next`.
אלו הרגיסטרים ש-`next` שמר כאשר הוא קרא
להחלפת הקשר בעבר.

```
jmp __switch_to
```

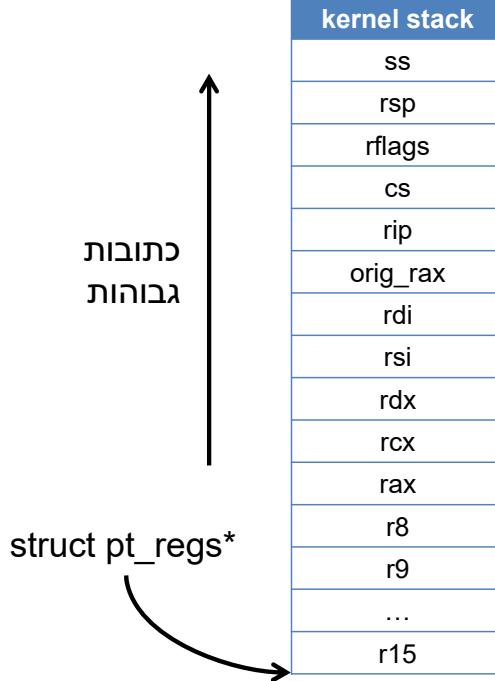
קפיצה (`jmp`) במקומות קרייה (`call`) לפונקציה. למה?
כתובת החזרה מהפונקציה (`__switch_to()`) כבר
שמורה על המחסנית של `next`.



Some registers are missing in the figure: r10, r11, rbx, rbp, r12, r13, r14 .

struct pt_regs

```
struct pt_regs {
    unsigned long r15;
    unsigned long r14;
    unsigned long r13;
    unsigned long r12;
    unsigned long rbp;
    unsigned long rbx;
    unsigned long r11;
    unsigned long r10;
    unsigned long r9;
    unsigned long r8;
    ...
    unsigned long rip;
    unsigned long cs;
    unsigned long flags;
    unsigned long rsp;
    unsigned long ss;
};
```



הו אטיפוס רשומה המכילה את ערכי הרגיסטרים בדיק בסדר בו הם מאוחסנים במחסנית לאחר קריית מערכת/קבלת פסיקה + ביצוע המאקרו `PUSH_REGS`.

Code from:

<https://elixir.bootlin.com/linux/v4.15.18/source/arch/x86/include/asm/ptrace.h#L54>

struct inactive_task_frame

```
struct inactive_task_frame {  
    unsigned long r15;  
    unsigned long r14;  
    unsigned long r13;  
    unsigned long r12;  
    unsigned long rbx;  
    unsigned long rbp;  
    unsigned long ret_addr;  
};
```

struct inactive_task_frame*



הוא טיפוס רשומה המכילה את ערכי הרגיסטרים struct inactive_task_frame __switch_to_asm() . בדיק בסדר בו הם מוחסנים במחסנית לאחר קריאת לפונקציה () .

Code from:

https://elixir.bootlin.com/linux/v4.15.18/source/arch/x86/include/asm/switch_to.h#L45

שלבי הפקנץיה (copy_thread())

```
int copy_thread(..., struct task_struct * p, {
```

מצבייע ל-PCB של תהליך הבן

- מוצאת את מבנה pt_regs אצל תהליך הבן:

```
struct pt_regs* childregs = task_pt_regs(p);
```

- מוצאת את מבנה pt_regs אצל תהליך האב:

```
struct pt_regs* regs = task_pt_regs(current);
```

- מעתקה את הרגיסטרים של האב לבן:

```
*childregs = *regs;
```

Code from: arch/x86/kernel/process_64.c

https://elixir.bootlin.com/linux/v4.15.18/source/arch/x86/kernel/process_64.c#L267

```
struct pt_regs* task_pt_regs(p) {
    unsigned long stack_top = p->task;
    unsigned long stack_bottom= ((unsigned long)stack_top) + THREAD_SIZE;
    return ((struct pt_regs*)stack_bottom) - 1;
}
```

שלבי הפונקציה `copy_thread()`

- מעדכנת את ערכו של `rax` ל-0 בתהילן הבן:

```
childregs->rax = 0; למה?
```

- מעדכנת את כתובת החזרה השמורה על המחסנית:

```
struct inactive_task_frame* frame =
    (struct inactive_task_frame*) (childregs) - 1;
frame->ret_addr = (unsigned long) ret_from_fork;
```

- מצביעה את `p->thread.rsp` לראש מחסנית הגרעין של הבן:

```
p->thread.rsp = (unsigned long) frame;
```

Answer: remember that `fork()` returns different values to the parent and the child; specifically, `fork()` return 0 to the child.

הפונקציה `ret_from_fork()`

- פונקציה זו מופעלת כאשר תהליך הבן מזומן לראשונה למעבד במהלך החלפת הקשר.

```
ret_from_fork:
```

```
    ...
    POP_REGS
    sysret
```

- המאקרו `POP_REGS` שולף את כל הרגיסטרים מהמחסנית.
- אחר כרך פקודת המכונה `sysret` קופצת חזרה לקוד המשתמש.
- למעשה, ביצוע הקוד ב-`ret_from_fork` יגרום לסיום הקראיה (`fork()`) בתהליך הבן עם ערך מוחזר 0.

סיום תרגיל בLINQ

או: איך מוממשת קריית המערכת `exit`?

דיון בכיתה – איך היitem ממשים את `exit` – מה עליה לעשות?



סיום ביצוע תהלייך

- תהלייך מסיים את ביצוע הקוד ע"י קריית המערכת (exit).
- גם אם קוד התהלייך לא קורא במפורש ל-exit, מתבצעת קרייה אוטומטית ל-exit לאחר שהפונקציה main חזרה:

```
int __libc_start_main(...){  
    .....  
    exit(main(...));  
}
```

- ביצוע קוד תהלייך יכול גם להיקטע בעקבות אירועים נוספים.
- תקלה לא מטופלת במהלך ביצוע הקוד, כגון גישה לא חוקית לזיכרון או חלוקה ב-0 – פרטים נוספים בתרגול על פסיקות.
- הריגת תהלייך אחד על-ידי תהלייך אחר – פרטים נוספים בתרגול על סיגנלים.



הפונקציה (`do_exit()`)

- הפונקציה (`do_exit()`) מופעלת בכל מקרה של סיום תהליך:
 - .1. משחררת את המשאבים שמשימוש התהליך: סגירת קבצים פתוחים, משחררת איזורי זיכרון, וכו'.
 - .2. רושמת את ערך הסיום של (`exit()` לשדה `exit_code` ב-`PCB`).
 - .3. מעדכנת קשרי משפחה: כל בניו של התהליך ששסיהם הופכים להיות בניים של `it`ו.
 - .4. משנה את מצב התהליך ל-`ZOMBIE_TASK`.
 - .5. קוראת לפונקציה (`schedule()`), אשר מוציאה את התהליך מתור הריצה וזמינה לריצה תהליך אחר במקוםו. ריצת התהליך מסתיימת סופית בפונקציה (`asm_to_asm_switch()`).

קובץ גרעין `kernel/exit.c`



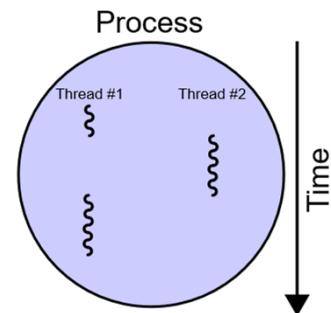
מחיקת PCB של הבן

- PCB של הבן מפונה רק כאשר תהליך האב מקבל חיוי על סיום התהיליך, באמצעות קריית מערכת כדוגמת `wait()`.
- מחיקת PCB מבוצעת ע"י הפונקציה **release_task()** שנקראת מתוך המימוש של `wait()`.
- פונקציה זו, בין השאר, מנתקת את התהיליך מרשותה התהיליכים הגלובאלית ומטבלת הערבול $\text{PCB} \rightarrow \text{PID}$, ומפנה את השיטה המוקצת ל-PCB ומחסנית הגרעין.
- **שאלה:** מי מבצע `wait` על תהליכיים יתומים?

תשובה: התהיליך `init` (מוגדר בקובץ `gruin.c`kernel/exit.c).

תרגול 7

חותמים (threads) בלינוקס
תמייצת גרעין לינוקס בחוטים
תכונות מקבילי באמצעות חוטים
מנגנוני סינכרון: מנעולים



TL;DR

- מעבדים מודרניים הם **מרובי ליבות**. איך אפשר לנצל אותם כדי לשפר ביצועים? לדוגמה, מיוון מערך גדול ע"י: חלוקה לשניים, מיוון כל חצי מערך בנפרד, ולבסוף מיזוג.
- **תהליכיים** לא משתפים זיכרון, ולכן הם פחות מתאימים לתוכנות מקבילים.
- **חותמים** (threads), לעומת זאת, פועלם **במרחב זיכרון משותף**.
- חוט הוא ייחידת ביצוע עצמאית בתוך תחילה ("Lightweight process").
- כל תחילה יכול להכיל מספר חוטים שיירצו במקביל.
- אבל שיתוף זיכרון בין חוטים יוצר גם בעיות, שאחת הנפוצות בהן היא: **היעדר אוטומיות בגישה למשתנים משותפים**.
- נלמד איך להתגבר על הבעיה באמצעות מנעולים (mutex/spinlock).

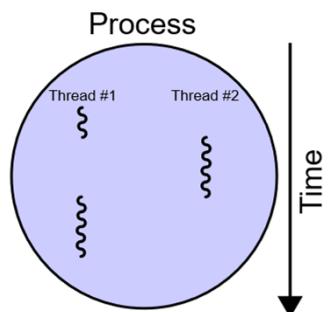
חותים (THREADS בLINUKO)

אנטי-דוגמה: מיזן מקבילי של מערך

```
void quick_sort(int a[], int length);  
  
int* parallel_sort(int a[], int length) {  
    int* b = (int*) malloc(length * sizeof(int));  
    pid_t p = fork();  
    if (p == 0) { // son  
        quick_sort(a, N / 2);  
    } else { // father  
        quick_sort(a + N / 2, N / 2);  
        wait(NULL);  
        // now merge the two subarrays into b  
    }  
    return b;  
}
```

למה המימוש הזה לא יעבוד?

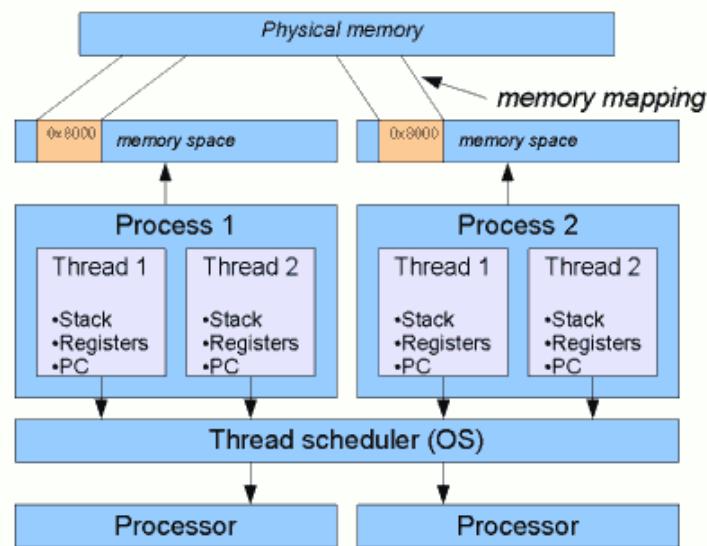
תשובה: לאחר `fork()` תחיליך הבן מקבל עותק נפרד של תחיליך האב, ולכן תחיליך האב לא ייראה את חצי המערך הממויין אלא את הערך המקורי של $a[0, \dots, N/2-1]$.



חותמים (threads)

- חוט הוא יחידת ביצוע בתחום תהליכי.
- באנגלית נקרא גם: Lightweight Process.
- בעברית נקרא גם: תהליכיון.
- תהליכי בלינוקס יכול לכלול מספר חוטים המשתפים ביניהם את כל משאבי התהליכי: מרחב הזיכרון, גישה לקבצים והתקני חומרה, ועוד.
- למרות ש热线 הוא רכיב של תהליכי, כל חוט הוא יחידת ביצוע עצמאית שנitin להריצ' על מעבד ללא קשר לשאר החוטים.
- כל חוט מהווה הקשר ביצוע נפרד – לכל חוט מחסנית ורגיסטרים משלו.
- ה-scheduler מזמן לריצה חוטים ולא תהליכי.

חותמים מול תהליכיים



From: https://www.javamex.com/tutorials/threads/how_threads_work.shtml



חותמים יקרים לשפר ביצועים

- במערכות מרובת מעבדים: כל חוט יפתר חלק מהמשימה של אותו תהיליך, והחותמים יריצו במקביל על מעבדים שונים.
- אבל זה דורש לפרק את הבעיה לתתי-בעיות בלתי תלויות – משימה לא טריומיאלית למכננה...
- גם במחשב עם מעבד יחיד ניתן לשפר ביצועים באמצעות שימוש בריבוי חותמים: חוט אחד יכול לוויטר על המעבד (למשל כדי להמתין לננטונים מהרשף) וחוט אחר ימשיך בביצוע חלק אחר של התוכנית.
- יתרון נוסף לחותמים: יצירת חוט זולה מעט מיצירת תהיליך חדש, מפני שהיא כרוכה בהעתיקת משאבים כמו טבלת הדפים וטבלת הקבצים הפתוחים.

מתי כדאי לא להשתמש בחוטים?

מתי לא כדאי?

- תוכניות קטנות ו פשוטות
עלולות לסייע מתקורה
מיותרת:
עלות ייצור חוטים חדשים.
- תהליכי חישובים במערכת
מעבד יחיד עלולים לסייע
מתקורה מיותרת:
עלות החלפות הקשר.

מתי כדאי?

- תהליכי חישובים "כבדים"
(למשל חישוי מגז האוויר)
יכולים לנצל יותר ליבות כדי
לשפר את הביצועים.
- חוטים יכולים להריץ משימות
בלתי תלויות, למשל הדפסת
מסמך במקביל לערכתו.
- אפליקציות שרת יכולות ליצור
חוט חדש לכל בקשה כדי לטפל
במספר בקשות בו-זמנית.

שימוש לב: בLINQ כל חוט מקבל time slice מסוון, אבל החוטים מחולקים ביניהם את זה- time slice של התהיליך כלו כדי שhotots לא יוכל לגנוב זמן מהתהיליכים אחרים במערכת.

תקשרות בין חוטים

- חוטים צריכים בדרך כלל לתקשר ולהחליף ביניהם מידע. אמצעי התקשרות הוא פשוט ביותר: קריאה וכתיבה למשתנים משותפים.
- המשתנים המשותפים צריכים להיות **גLOBליים** או להיות **moveרים** **FROM** **TO**.
- משתנים משותפים הם גם חסרים: יש לאמת את הגישות אליהם על- מנת למנוע את שיבוש הנתונים.
- ישנו מצבים שבהם חוטים לא נדרשים לשתף ביניהם מידע.
 - לדוגמה: חיפוש מילה ספציפית במספר גדול של קבצים. כל חוט יקרא קובץ וידפיס למסך את המופיעים של המילה בקובץ שהוא בדק.
 - בעיות שלא דורשות תקשורת בין החוטים נקראות **parallel** **embarrassingly**.

ספריית **pthreads**

- בשנת 1995 הוציא **תיקן POSIX Threads** (POSIX Threads), המגדיר אוסף טיפוסים ופונקציות המאפשרים עבודה עם חוטים במערכות **Unix**.

הטיפוסים והפונקציות מוגדרים בקובץ `pthread.h`, המוממשים בספרייה **pthreads** (ספרייה משתמש כמו `libc`).

- כדי להשתמש בספרייה **pthreads** יש:
 - להויסיף קובץ header בתחילת קוד C:

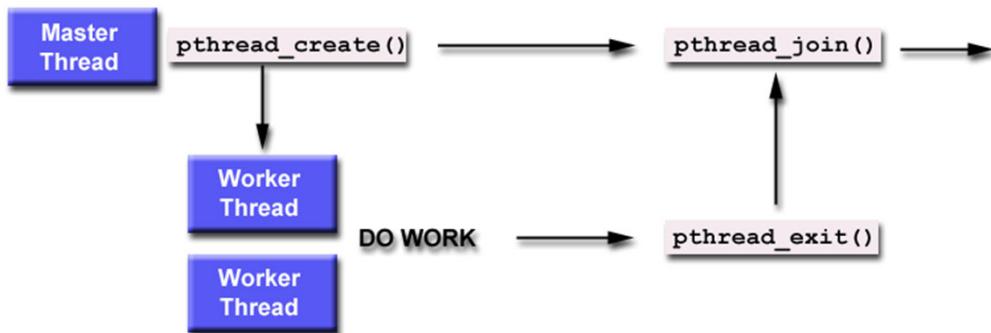
```
#include <pthread.h>
```

- להויסיף את הדגל `-pthread` במהלך הידור:

```
gcc myprog.c -pthread -o myprog
```

- הדגל מגדיר מספר macros ומקשר את התוכנית עם הספרייה הנחוצה.

סכמת עבודה



- שימוש לבב: שילוב בין תהליכי לוחטים הוא אפשרי אך אינו מומלץ.

Figure from: <https://computing.llnl.gov/tutorials/pthreads/>

Read more at: <http://www.linuxprogrammingblog.com/threads-and-fork-think-twice-before-using-them>

דוגמא ליצירת חוטים

```
#include <stdio.h>
#include <unistd.h>
#include <pthread.h>

void* print_m(void* ptr) {
    char* m = (char*)ptr; // cast the thread argument
    printf("%s PID = %d, pthread ID = %ld\n",
           m, getpid(), pthread_self());
    return NULL;
}

int main() {
    pthread_t t1;
    const char* m = "I'm Thread 1!";
    pthread_create(&t1, NULL, print_m, (void*)m);
    // we assume pthread_create() didn't fail
    pthread_join(t1, NULL);
    return 0;
}
```

מה ידפס הקוד הבא?

The program would print something like:

I'm Thread 1! PID = X, pthread ID = T

יצירת חוט חדש

```
int pthread_create(pthread_t *thread,  
                    pthread_attr_t *attr,  
                    void* (*start_routine)(void*) ,  
                    void *arg);
```

- פעולה: יוצרת חוט חדש המתבצע במקביל לחוט הקורא בתוך אותו תחילך. החוט החדש מתחילה לבצע את הפונקציה המופיעה בפרמטר `start_routine` וממת בסיום ביצוע הפונקציה.
- ערך מוחזר:
 - 0 במקרה של הצלחה. כמו כן, מזיהה החוט החדש נכתב למשתנה המוצבע ע"י `.thread`
 - ערך שגיאה שונה מ-0 במקרה של כישלון.

ערכים השגיאה האפשריים מופיעים ב-[page man](#).

יצירת חוט חדש

פרמטרים:

- **thread** – מצביע למקום בו יוחסן מזהה החוט החדש במקרה של סיום הפעקציה בהצלה.
- **attr** – מאפיינים המתאימים את תכונות החוט החדש, כגון האם החוט הוא חוט גרעין או חוט משתמש, האם ניתן לבצע לו חויב, כלומר להמתין לסיומו, וכו'. בד"כ נספק ערך **NULL** המציין חוט מערכת שנייה להמתין לסיומו.
- **void* (*start_routine) (void*)** – מצביע לfuunkcia שתהווה את קוד החוט. הערך המוחזר מfuunkcia זו במקרה של סיום הבלתי הימן **ערך הסיום של החוט**.
- **arg** – פרמטר שישופק לfuunkcia עם הפעלה.

קבלת מזהה החוט

```
pthread_t pthread_self();
```

- **פעולה:** מחזירה לחוט הקורא את המזהה של עצמו. מזהה זה הוא פנימי לסדרייה pthreads **ואינו קשור** ל-PID של החוט.
- **מתוך ה-man page**:
“Thread identifiers should be considered opaque: any attempt to use a thread ID other than in pthreads calls is nonportable and can lead to unspecified results.”

סיום חוט

```
void pthread_exit(void *retval);
```

- פעולה: מסיימת את פעולת החוט הקורא. ערך הסיום יוחזר לחוט שימתיין לסיום חוט זה.
- פרמטרים:
retval – ערך סיום (בדומה לזה של exit).

הriegת חוט

```
int pthread_cancel(pthread_t thread);
```

- **פעולה:** מסיימת את ביצוע החוט `thread` בעזרת סיגנל `SIGABRT`.
- ערך סיום הביצוע של החוט שנהרג יהיה `PTHREAD_CANCELED`.
- **פרמטרים:**
 - `thread` – מזהה החוט המיועד לסיום.
- **ערך מוחזר:**
 - 0 במקרה של הצלחה.
 - ערך שגיאה שונה מ-0 במקרה של כישלון.

המתנה לסיום חוט

```
int pthread_join(pthread_t thread,  
void **thread_return);
```

- **פעולה:** גורמת לחוט הקורא להמתין לסיום החוט המזווה ע"י `.thread`.
- ניתן להמתין על סיום אותו חוט פעם אחת לכל היוטר – ביצוע (`join`) `pthread_join()` על אותו חוט יותר מפעם אחת ייכשל.
- כל חוט יכול להמתין לסיום כל חוט אחר באותו תהליך.
- המתנה על סיום החוט משחררת את מידע הניהול של החוט בرمת הספרייה `pthreads` וברמת הגראן.

המתנה לסיום חוט

• פרמטרים:

- `thread` – מזהה החוט שמתכוון לסיומו.
- לא ניתן להמתין ל"סיום חוט כלשהו" בדומה ל-`wait()`.
- `thread_return` – מצביע למקום בו יוחסן **ערך הסיום** של החוט עבורו ממתיינים.
- ניתן לציין `NULL` כדי להתעלם מערך הסיום.
- **ערך מוחזר:** 0 במקרה של הצלחה, וערך שונה מ-0 במקרה כישלון. כמו כן, במקרה של הצלחה ערך הסיום נכתב למשתנה המוצבע ע"י `thread_return` (אם אינם `NULL`).

סיום חוטים ותהליכיים

- חוט יכול להסתיים במספר דרכים שונות:

אם התהיליך מסתיים?	אם החוט מסתיים?	סיבת הסיום
לא בהכרח (רק אם החוט שהסתיים היה האחרון)	כן	קריאה ל- <code>(pthread_exit()</code> בטור קוד החוט
		קריאה ל- <code>(pthread_cancel()</code> מחוץ אחר
		חרזה מהפונקציה <code>start_routine</code> (הפונקציה המבצעת של החוט)
		קריאה לקריאה מערכת <code>(exit())</code> ע"י חוט כלשהו בקבוצה של החוט המדובר
		פעולה לא חוקית באחד החוטים (למשל, חילוקה באפס)
		חרזה מהפונקציה <code>(main)</code> של החוט הראשי (שקלול לקריאה ל- <code>(exit())</code>)

אם החוט הראשי רוצה להסתיים מבליל להרוג את התהיליך כולו, הוא יכול לקרוא ל- `.pthread_exit()`

תמיכת גרעין לינוקס בחוטים

חוטים בגרעין לינוקס

- בגרעין לינוקס, חוטים ממומשים למעשה כתהליים רגילים המשתפים ביניהם משאבים כגון זיכרון, גישה לקבצים וחומרה.
- כל תהלייר נוצר עם חוט יחיד – **החוט הראשי** (primary thread) באמצעות קריית המערכת (`fork()`).
- חוטים נוספים נוצרים באמצעות **קריית המערכת** (`clone()`)
 - קריית מערכת זו היא הבסיס לתמייה בחוטים.
- בנגד לתהליים, אין קשר משפחתי בין החוטים.
 - אין חוט אב וחתוט בן.
 - כל חוט יכול להממש לסיום של חוט אחר כלשהו של אותו תהלייר.
 - כל חוט יכול להרוג חוט אחר כלשהו של אותו תהלייר.

קובוצת חוטים (thread group)

- לכל חוט, בהיותו תהיליך רגיל, יש PCB משלהו ו-ID **PID** משלהו.
- עם זאת, המתכונת מצפה שלכל החוטים השبيיכים לאותו תהיליך ניתן יהיה להתייחס דרך PID יחיד – של התהיליך המכיל אותם.
- פועלות ()`getpid()` תחזיר את אותו PID בכל החוטים של אותו תהיליך.
- קריאות מערכות כמו ()`kill()` הפעולות על ה-ID של התהיליך צריכות להשפייע על כל החוטים בתהיליך.
- לכן, ניתן מחדדת את כל החוטים של תהיליך מסוים **לקבוצת חוטים (thread group)** כדי שאפשר יהיה להתייחס אליהם יחד.

קובוצת חוטים (thread group)

- השדה **tgid** במתאר התהיליך מכיל את ה-PID המשותף לכל החוטים באותו קבוצה.
 - למעשה, זהו ערך החוט הראשוני (הראשי) של התהיליך.
 - חוטים חדשים יקבלו ערך PID חדש וערך TGID זהה לחוט הראשוני.
 - קריאה המרצת `getpid()` ממחזירה למעשה את `tgid`.
- השדה **kgroup** במתאר התהיליך (`task_struct`) הוא ראש הרשימה המקושרת של כל החוטים באותו קבוצה.
- פעולות על ה-PID המשותף מתורגםות לפעולה על קבוצת החוטים המתאימה לו-PID.
 - למשל: `(kill(pid, SIGKILL).tid==pid)` הורגת את כל החוטים עבורם `pid`.

סיכום: PID מול TID

- המשתמש ומערכות הפעלה מסתכלים על חוטים באופן שונה:

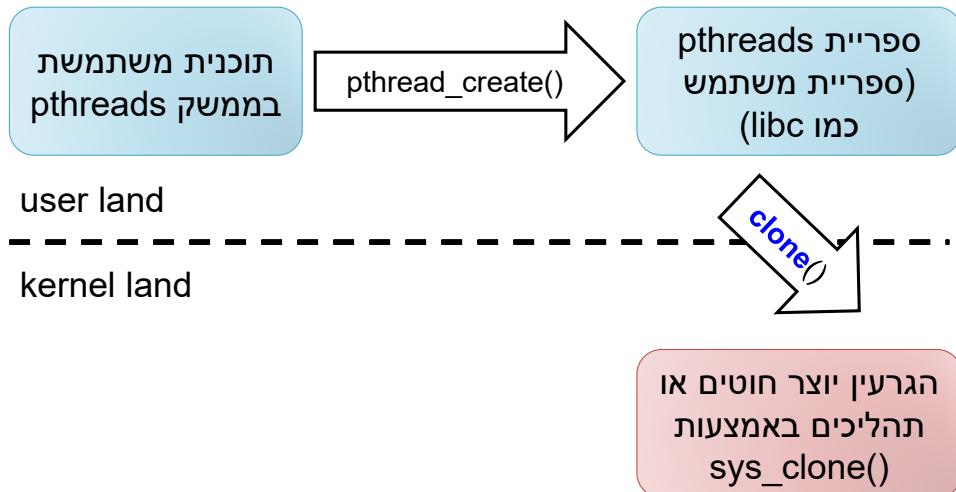
- המשתמש:**

- תהליך מרובה חוטים מורכב מחוט ראי' וחוטים שונים המרוכזים תחת PID יחיד (שהוא בעצם גם ה-TID).
- ניתן לקבל PID זה באמצעות (`getpid()`).
- לכל חוט מזיהה נוסף הנקרא TID או ID Thread ID (זהו בעצם ה-PID האמתי של החוט).
- ניתן לקבל TID זה באמצעות (`gettid()`).

- מערכות הפעלה:**

- מסתכלת על כל חוט וחותט כתהליך נפרד, בעל PID מסויל עצמו. קיימת "קישורת גורלוות" של כל התהליכים המרוכזים תחת אותו TGID.

תרשים: חוטים במבט מערכתי



קריאה המערכת clone()

```
int clone(int (*fn)(void*), void *child_stack,  
          int flags, void *arg);
```

- פעולה: יוצרת תהליך בן המשותף עם תהליך האב משאים ונתונים לפי בחירה.

- פרמטרים:

- fn – מצביע לפונקציה שתהוו את הקוד הראשי של התהליך החדש.
- arg – הפרמטר המועבר לפונקציה ()fn בתחילת ביצוע התהליך החדש.
- כשביצוע הפונקציה fn(arg) מסתיים, נגמר התהליך החדש.
- child_stack – מצביע לראש המחסנית של התהליך החדש. **איזה מחסנית?**
- **תשוכנות:** המחסנית גדרה לכיוון הכתובות הנמוכות.
- **משתמש / גרעין?**
- לכן child_stack צריך מצביע לסוף בלוק הזיכרון המוקצה לטובות המחסנית.

תשובה: מחסנית משתמש כموן – קראת המערכת נקראת מקוד משתמש ואסור לתת לה אפשרות לשלווט במיקום מחסנית הגרען.

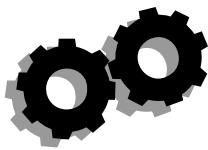
קריאה המערכת (clone)

- מסכת דגלים הקובעת את צורת השיתוף בין התהילך הקורא והתהילך החדש. להלן מספר דגלים אופייניים:

שיתוף מרחב הזיכרון	CLONE_VM
שיתוף טבלת הקבצים הפתוחים	CLONE_FILES
שיתוף טבלת נתוניםעובדת עם קבצים, המכילה נתונים כגון ספריית העבודה הנוכחיית ועוד	CLONE_FS
لتהילך החדש יהיה אותו אב כמו התהילך הקורא (אחרת החדש יהיה הבן של הקורא)	CLONE_PARENT
התהילך החדש הוא חוט באוותה קבוצת חוטים כמו התהילך הקורא (אותו pid). גורר גם CLONE_PARENT	CLONE_THREAD

- ניתן לשלב מספר דגלים יחד באמצעות OR לוגי בינויהם, לדוגמה:
CLONE_FS | CLONE_VM

- **ערך מוחזר**: במקרה של הצלחה מוחזר ה-PID של התהילך החדש, אחרת -1.



מימוש קריית המערכת (clone)

- בתוך הגרעין, sys_clone() משתמש שירות בפונקציה do_fork() עליה למדנו בתרגולים קודמים, ומעבירה לה את הדגלים על-מנת לקבוע לכל משאב אם לשתף אותו או ליצור אותו מחדש.

הערה למתקדמים: החתימה של sys_clone() שונה מאוד מהחתימה של clone() שראה המשתמש:

```
asmlinkage int sys_clone(struct pt_regs regs);
```

זה כמובן לא בעייתי, כי במקרה זה הגרעין מסתכל על כל המבנה pt_regs ולא רק על הפרמטרים שהעביר לו המשתמש.

הפרק

Multithreaded programming



תכנות מקבילי באמצעות חוטים

וגם: מה קורה בגין לא מתואמת למשתנים משותפים?

תכנית

לדוגמה

הfonקציה (f) תרוץ 1000 פעמים במקביל, ובכל פעם תמלא איבר נוסף של המערך.

מה יהיה ערכו של a[999] בסיום התוכנית?

```
#include <pthread.h>
#include <stdio.h>
#define N 1000
int i = 0;
int a[N];

void* f(void* arg) {
    a[i] = i;
    i++;
    return NULL;
}

int main() {
    pthread_t threads[N];
    for (unsigned int i=0; i<N; i++)
        pthread_create(&threads[i], NULL, f, NULL);
    for (unsigned int i=0; i<N; i++)
        pthread_join(threads[i], NULL);
    printf("a[999] = %d\n", a[999]);
    return 0;
}
```

פלט התוכנית לדוגמה

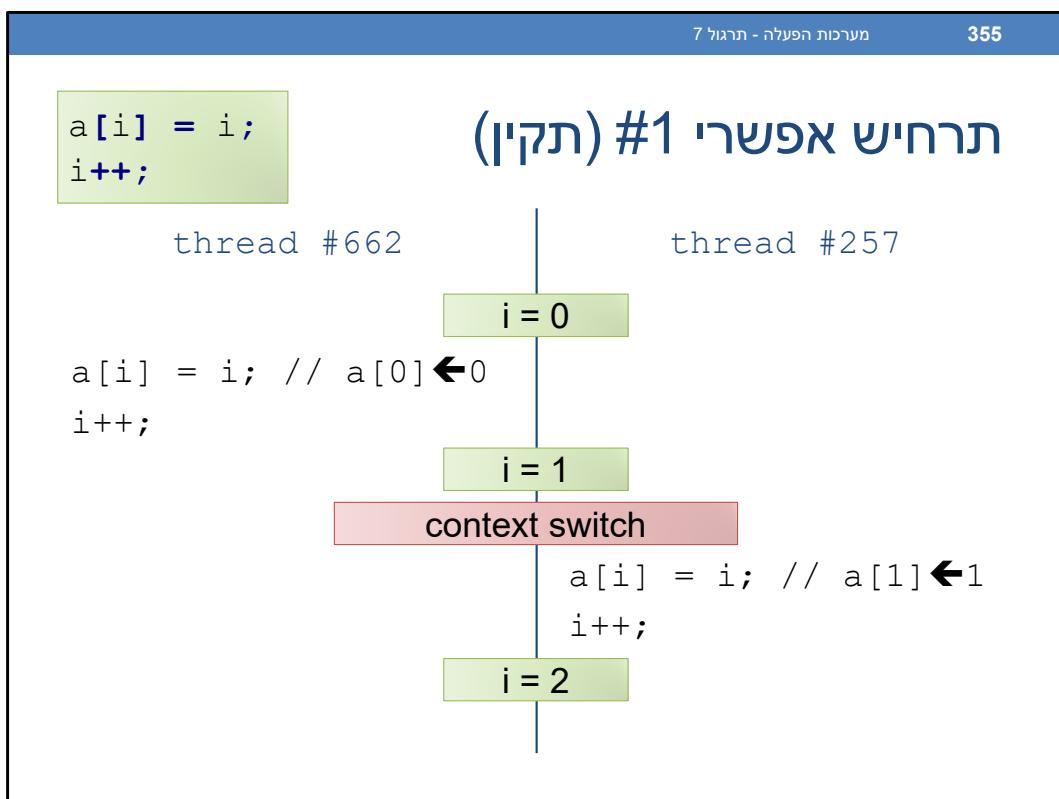
```
>> gcc -pthread -O2 main.c
>> ./a.out
a[999] = 999
>> ./a.out
a[999] = 999
>> ./a.out
a[999] = 0
```

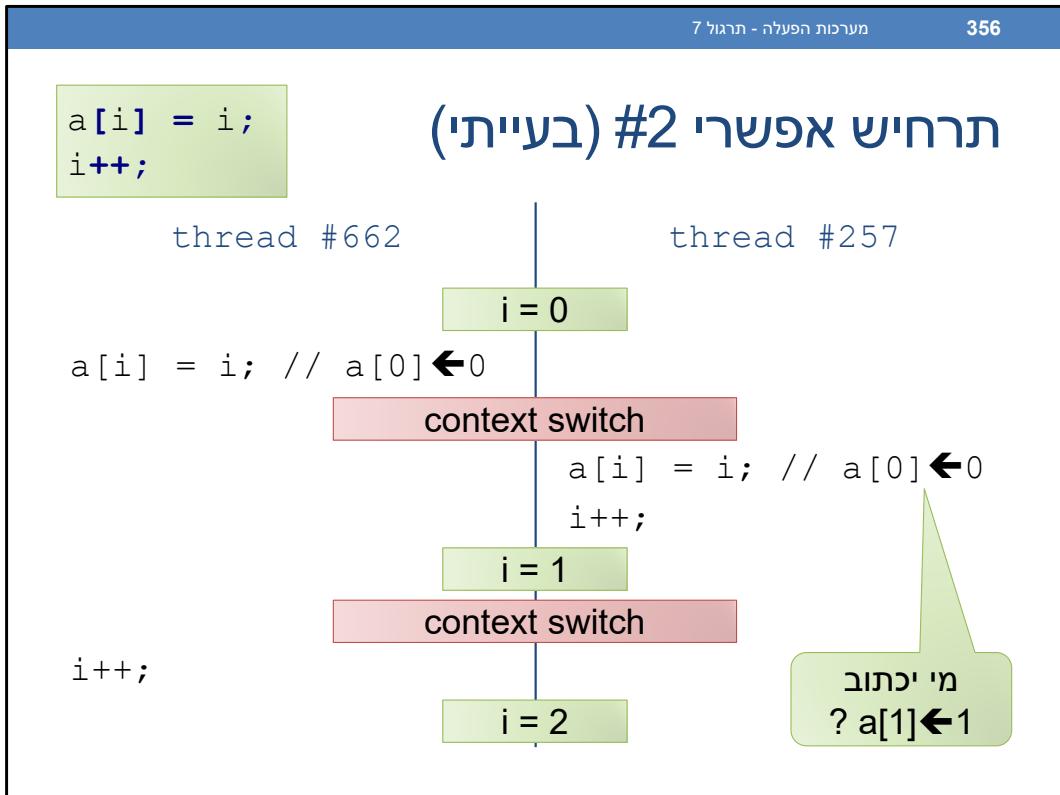


0 הוא הערך המקורי של איברי המערך []*a* בגלל שהוא משתנה גלובלי.

מה קורה כאן?

- החלפת הקשר בין החוטים יכולה לגרום בכלל נקודת זמן, בפרט באמצעות הפונקציה של חוט מסוים.
- פלט התכנית תלוי בזמןן של החוטים ובסדר בו הם מתבצעים – מצב שנקרא **race condition** (תנאי מרוץ).
- מכיוון שאנו לא שולטים בזמןן של תהליכיים (או חוטים), תוכנית המפעילת race condition נחשבת תקולה (buggy).
- ליתר דיוק, תוכנית צאת היא בעלת התנהגות לא מוגדרת.
- קשה לדבג תוכניות כאלה כי קשה לשחזר את ההתנהגות הביעיתית.





התרחיש המתואר כאן אכן עשייתי, כי אף חוט לא יכתוב את הערך 1-[1]א, אבל הוא לא מסביר מדויק [999]א יכול להישאר בערך 0.

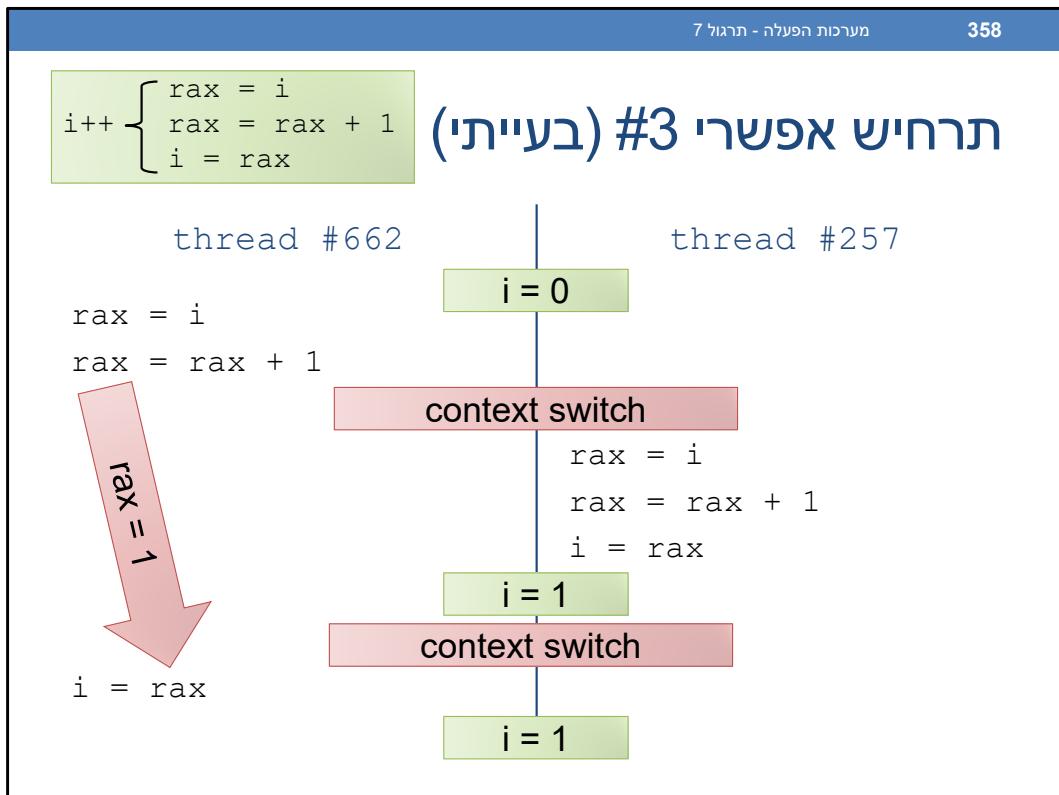
דges: במערכת מרובת מעבדים התרחיש יכול לקשר גם ללא החלטת הקשר אם שני החוטים ירצו במקביל על מעבדים שונים.

בשורת C אחת race condition

- שימושו לב: החלפת הקשר יכולה לגרום גם באמצע שורת C, כי הקומpileר עשוי לתרגם שורת C אחת למספר פקודות אסמבלי.
- לכן, race condition יכול לגרום גם במקומות לא צפויים.
- לדוגמה, הקומpileר יכול להדר את השורה `++i` לקוד האסמבלי הבא:

f:

```
...  
    movl (%rbx), %rax  
    addl $1, %rax  
    movl %rax, (%rbx) }  
...  
  
פואודו-קוד:  
    rax = i  
    rax = rax + 1  
    i = rax
```



התרחיש הזה מדגים מדוע [999]א יכול להישאר בערך 0. (שים לב כי 0 הוא הערך ההתחלתי של [999]א בגלל שהוא משתנה גלובלי). גם כאן התרחיש יכול לקרות ללא החלפת הקשר אם שני החוטים ירצו במקביל על מעבדים שונים.

קטע קרייטי

- קטע קוד הניגש למשאב משותף (למשל משתנה בזיכרון).
- ביצוע קטע קרייטי במקביל עלול לגרום להתנגשות לא רצiosa.
- **תכנית תקינה צריכה להבטיח מניעת הדדיות – לכל היוטר חוט אחד ירץ את הקטע הקרייטי בכל רגע נתון.**
- במקרה אחריות הפקודות בקטע הקרייטי צריכה להיות אטומית ביחס לגישות אחירות למשאב המשותף – או שכל הפקודות בקטע הקרייטי ירצו יחד יוסתיימו, או שהן לא ירצו כל.

```
void* f(void* arg) {  
    a[i] = i; }  
    i++; }  
    return NULL;  
}
```

בדוגמה הקודמת: שתי השורות
האלו הן קטע קרייטי

מנגנוני סינכרון: מנעולים

בהמשך הפרק זהה מופיע מושג הקיפאון (deadlock).
הסטודנטים אמורים לראות אותו בהרצאה, אבל זה לא תמיד המצב...

מנעולים

- **מנעולים** הם אחד המנגנונים הבסיסיים לミימוש מניעה הדדית.



- האנלוגיה: קטע קרייטי \Leftrightarrow חדר עם דלת מוגנת ע"י מנעול עם מפתחה בפנים.
- כדי להיכנס לחדר (הקטע הקרייטי) צריך לנעול את המנעול ולשיטם את המפתח בדלת.
- ביציאה מהחדר יש לפתוח את המנעול ולהשאיר את המפתח בדלת (לטובת החוטים האחרים).

שימוש לב:

- בכל רגע נתון, לכל היוטר חוט אחד יכול לתפוס / להחזיק / לנעול את המנעול.
- רק החוט המחזק במנעול אמור לשחרר אותו (בעלות על המנעול).

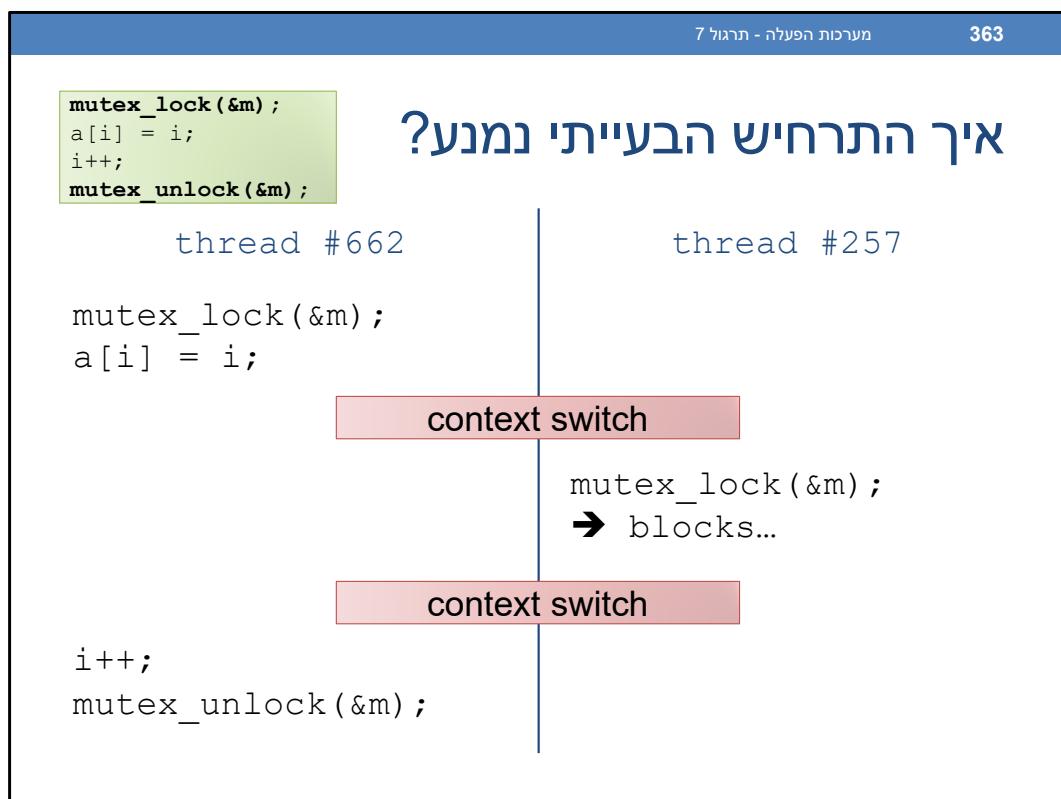
מנועלים ב-pthreads

- בתקן pthreads מנועלים נקראים **.mutex**.
- קיוצר של mutual exclusion (מניעה הדדית).

- נתן את הדוגמה הקודמת בעזרת נעילה:

```
pthread_mutex_t m;  
  
void* f(void* arg) {  
    pthread_mutex_lock(&m);  
    a[i] = i;  
    i++;  
    pthread_mutex_unlock(&m);  
    return NULL;  
}
```

- אם המנועול אינו נעול, החוט נועל אותו ונכנס לקטע הקרייטי.
- אם המנועול כבר נעול, החוט נחסם עד אשר המנועול ישוחרר.



דוגמת קוד עם mutex

```
// shared var
// must be protected
long long count;

pthread_mutex_t m;

void update_count() {
    pthread_mutex_lock(&m);
    count = count * 5 + 2;
    pthread_mutex_unlock(&m);
}

long long get_count() {
    long long c;
    pthread_mutex_lock(&m);
    c = count;
    pthread_mutex_unlock(&m);
    return c;
}
```

- **שאלה:** מדוע צריך להגן על הגישה ל-
?update_count() בטור count
- **תשובה:** כדי למנוע שיבוש ערך count
בעדכנים מוחוטים שונים.
- **שאלה:** מדוע צריך להגן על הגישה ל-
?get_count() בטור count
- **תשובה:** כדי למנוע קבלת תוצאות
חלקיים הנוצרות במהלך העడכן.
- **שאלה:** נניח שפעולות העידקון הייתה
+ count++. האם עדין צריך להגן על
הfonקציה() update מנעול ?
- **תשובה:** כן, כי לא מובטח שהקוד הנפרש
בasmmbil הינו אטומי.

שאלות נוספת:

1. למה לא משתמשים בשני מניעולים נפרדים בפונקציות update/get ?
כי אחרת הייתה מניעה הדדית רק בין קוראים ורק בין כתבים .
2. מה היה קורה אם המניעול היה משתנה מקומי ?
לא היה סינכרון כי זה משתנה לוקלי והוא שונה בין חוטים שונים .

מנועלים עם/בלי המתנה

spinlock

- כאשר חוט מנסה לתפוס מנעול נועל, הוא בודק את ערך המנעול שוב ושוב מבלי **לצאת מתחור הריצה**.
- טכנית כזו נקראת גם: **.polling**, **.busy waiting**, **busy looping**
- **יתרונות**: התהיליך יכול להמשיך לרווח עוד בקבינטום הנוכחי, וכך לחסוך את התקורה על החלפת הקשר.
- עדיף כאשר זמן המתנה המשוער נמוך יותר מהמחיר של החלפת הקשר (כלומר, כאשר הקטע הקרייטי קצר, כפי שקרה לרוב בקוד גרעין).

mutex

- כאשר חוט מנסה לתפוס מנעול נועל, מערכת הפעלה תעביר אותו **لتור המתנה** ותבצע החלפת הקשר.
- כאשר המ נעול ישוחרר, מערכת הפעלה תעיר את אחד החוטים המحققים למנעול.
- **יתרונות**: התהיליך חדש יכול לרווח מיד, וכך לא מתבזבז זמן מעבד יקר.
- עדיף כאשר זמן המתנה המשוער גבוה יחסית (כלומר, כאשר הקטע הקרייטי ארוך, כפי שקרה לרוב בקוד משתמש).

אנלוגיה ל-spinlock: ילד שsspואל "כבר הגענו? ועכשו? ועכשו? ועכשו? ...".

אנלוגיה ל-mutex: אומרים לילד לлечת לישון, ומבטיחים לו שנעיר אותו כאשר הגענו.

הערה: גם חוט שמנסה לתפוס spinlock יחולף לבסוף בתהיליך אחר, כאשר יסימם את פיסת הזמן שהוקצתה לו.



אתחול מנעול mutex

```
#include <pthread.h>
int pthread_mutex_init(
    pthread_mutex_t *mutex,
    const pthread_mutex_attr_t *mutexattr);
```

פרמטרים:

- mutex – המנעול עליו מבוצעת הפעולה.
- mutexattr – מגדר את תכונות mutex.
- NULL – עבור סוג ברירת המחדל.
- ניתן לקרוא על סוגים נוספים ב-pages man.
- **ערך מוחזר: 0 בהצלחה, אחר כבישלו.**

To initialize an ERRORCHECK mutex:

```
pthread_mutexattr_t attr;
pthread_mutexattr_init(&attr);
pthread_mutexattr_settype(&attr,
PTHREAD_MUTEX_ERRORCHECK);
pthread_mutex_t error_check_mutex;
pthread_mutex_init(&error_check_mutex, &attr);
```



פעולות על מנעולי mutex

- נעילת mutex:

```
int pthread_mutex_lock(pthread_mutex_t *mutex) ;
```

- הפעולה חוסמת עד שה-mutex מתפנה ואז נעלת אותו.

- ניסיון לנעילת mutex:

```
int pthread_mutex_trylock(pthread_mutex_t *mutex) ;
```

- הפעולה נכשלת אם ה-mutex כבר נעול, אחרת נעלת אותו.

- שחרור mutex נעול:

```
int pthread_mutex_unlock(pthread_mutex_t *mutex) ;
```

- ניסיון לשחרר מנעול שאינו נעול תביא להתנהגות לא מוגדרת.

- פינוי mutex בתום השימוש:

```
int pthread_mutex_destroy(pthread_mutex_t *mutex) ;
```

- הפעולה נכשלת אם ה-mutex מואתחל אבל נעול.

שימוש תקין במנועלים

- בקוד תקין רק החוט שמחזיק במנועל הוא זה שמשחרר אותו.
- לעומת זאת, הפעולות הבאות יובילו להתנהגות לא מוגדרת:
 - .1. נעליה חוזרת ע"י החוט שהחזיק במנועל.
 - .2. שחרור המנועל ע"י חוט שאינומחזיק במנועל.
 - .3. שחרור מנועל שאינו נעול.
- דיבוג קוד עם מנועלים הוא מטגר יותר בגלל שהקוד כבר לא דטרמיניסטי – במקרה מהריצות יש באג, במקרה מהריצות אין באג.
- יש כלים שעוזרים בבעיה, לדוגמה helgrind או sanitizers.

פקודות מכונה אוטומיות

- פקודה מכונה אוטומית (atomic operation) היא פקודה המעדכנת את מצב המערכת (מעבד+זיכרון) בצורה אוטומית.
- ככלומר מעבדים אחרים יוכלים לראות מצבם ביןיהם של ביצוע הפקודה ואיןם יכולים לעדכן את מצב המערכת תוך כדי ביצוע הפקודה.
- במערכת מעבד יחיד: כל פקודה מכונה היא אוטומית מכיוון שהיא אינה יכולה להיקטע ע"י פסיקה.
- במערכת מרובת מעבדים: פקודה מכונה אוטומית חייבת לנעל את ערז הגישה של כל המעבדים האחרים לזכרון עד לסיוםה.
- הנעה מתבצעת באמצעות הוספה קידומת lock לפקודה.
- למשל: `x lock; inc` היא פקודה מכונה המבצעת `++x` בצורה אוטומית.

https://wiki.osdev.org/Atomic_operation

פקודות מכונה אוטומיות

- פקודות מכונה אוטומיות מגדירה למעשה קטע קרייטי באורך פקודה אחד, וכך מאפשרת להימנע משימוש במנועל.
- בנוסף, פקודות מכונה אוטומיות מאפשרות למש בצורה פשוטה יחסית אמצעי סינכרון כמו מנעולים.
- לדוגמה: ארכיטקטורת 64x מציעה את פקודה `bit test and BTS` (`set`), אשר מדיליקה בית מסויים ברגיסטר או כתובת בזיכרון ומחזירה את ערכו הקודם בדגל CF ברגיסטר הדגלים.
- פונקציית הגרעין (`test_and_set_bit`) משתמשת בפקודת המכונה `BTS` כדי להדילק משתנה ולהחזיר את ערכו הקודם בצורה אוטומית.

1

שאלה מבחן –
מימוש מניעולים

סעיף א'

```

typedef struct lock {
    bool is_locked;
} lock_t;

void init(lock_t* l) {
    l->is_locked = 0;
}

void lock(lock_t* l) {
    while (l->is_locked);
    l->is_locked = 1;
}

void unlock(lock_t* l) {
    l->is_locked = 0;
}

```

- להלן מימוש של מנעול ללא תמיכת הגרעין.
 - הניחו כי השמת ערך לשנתנה בזיכרון או קריאה שלו הין אוטומיות.
 - התעלמו מבעיות קוחרנטיות וקונסיסטנטיות.
- האם המימוש מבטיח מנעה הדדיות?
 - לא. ניתן דוגמה נגדית:
 - חוט A מנסה ל特派ס את המנעול, בדק את הערך של `is_locked` והוא נדיין 0 ← עובר את לולאת `while`.
 - בעת מתבצעת החלטת הקשר וחוט B בודק את הערך של `is_locked` והוא נדיין 0 גם הוא עובר את לולאת `while`.
 - שני חוטים בקטע הקרייטי בו זמני!

סעיף ב'

```
typedef struct lock {  
    bool is_locked;  
} lock_t;  
  
void init(lock_t* l) {  
    l->is_locked = 0;  
}  
  
void lock(lock_t* l) {  
    while (test_and_set_bit  
        (l->is_locked));  
}  
  
void unlock(lock_t* l) {  
    l->is_locked = 0;  
}
```

- כעת נתון מימוש בעזרת הפקודה האוטומית (`test_and_set_bit()`, המدلיקה בית ומוחזירה את ערכו הקודם.
- האם המימוש מבטיח מניעה הדדית?
 - כ. ההוכחה בעזרת נפנופי יד'ים...
- האם המימוש מונע הרעבה?
 - לא. ניתן דוגמה נגדית:
 - שני חוטים A,B רצים לסירוגין.
 - חות A תמיד תופס את המנגול לפני סיום הקווונטום שלו.
 - חות B תמיד מכלה את הקווונטום שלו בהמתנה למנגול.

העשרה (לא בחומר)

מבוא לתוכנות מקבילי

- במערכות הפעלה תואמות POSIX (כמו לינוקס ו-macOS של אפל) הספרייה pthreads הסטנדרטית היא pthreads.
- יתרונות: ותיקה מאוד, אמינה.
- חסרונות: לא מספקת אבסטרקציות נוחות ולן נחשבת יחסית קשה לתוכנות (נדרשות הרבה שורות קוד).
- TBB (Threading Building Blocks) של אינטל היא ספריית C++ בקוד פתוח לפיתוח קוד מקבילי.
 - יתרונות:
 - אבסטרקציות ברמה גבוהה (למשל `parallel_for`) שמקילות על המתכנת.
 - portability בין מערכות הפעלה שונות. (הימוש בלינוקס מתבסס על pthreads).
 - חסרונות: אין portability בין ארכיטקטורות שונות (למשל ARM).
- OpenMP היא הרחבה שפה של C,C++,Fortran שמנומשת ע"י הנחיות למזהדרם.
 - יתרונות:
 - אבסטרקציות ברמה גבוהה (למשל `parallel_for`) שמקילות על המתכנת.
 - portability בין מערכות הפעלה ארכיטקטורות שונות. (השימוש בלינוקס מתבסס על pthreads).
 - המבוי נכתב כתוספת לקוד הקיימ, קר שבעל רגע ניתן להדר את הקוד בגרסה לא מקבילת.
 - חסרונות:
 - נדרשת תמיכת קומpileר (למשל clang).

תרגול 8

מנגנוני סינכרון: משתני תנאי

מנגנוני סינכרון: סמפוריים

דוגמה: שימוש מנעול קוראים-כותבים

סינכרון בגרעין לינוקס

TL;DR

- בתרגול הקודם למדנו לכתוב קוד מקבילי באמצעות חוטים.
- ראיינו שבכל בעיה לא טריוויאלית יש צורך בסyncronization בין החוטים.

- היום נלמד על מנגנוני סינכרון נוספים של ממשק pthreads :

להבטחת אוטומטיות סדר	
משתני תנאי (condition variables)	מנעולים (mutexes)
סמלולי בקרה שרצים במקביל	

- לבסוף, נלמד דוגמה נוספת של קוד מקבילי חשוב: גרעין לינוקס.
- קוד הגרעין לא משתמש בחוטים, אבל **יגש לזכרון משותף** מתוך מספר **סמלולי בקרה שרצים במקביל**, ולכן העקרונות של מנגנון תקפים גם עכשו.

ממשק pthreads מספק גם מנגנוני סינכרון נוספים ומתחכמים יותר שנראה בתרגול זה ובשיעור הבית, לדוגמה: ... barriers, reader-writer locks, ...

Question: How are the pthreads primitives (mutex, condition variables, ...) implemented?

Answer: LinuxThreads uses user-space signals (specifically SIGUSR1, SIGUSR2), whereas NPTL leverages the futex() system call, which was added on kernel version 2.6 and allows processes to add/remove themselves to/from a kernel wait queue.

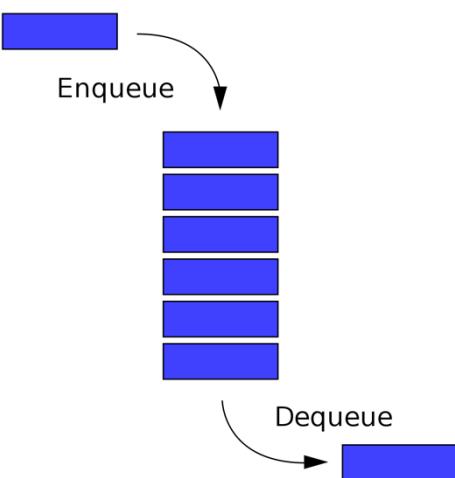
Read more at:

<https://en.wikipedia.org/wiki/Futex>

<https://stackoverflow.com/questions/45764378/how-are-threads-processes-parked-and-woken-in-linux-prior-to-futex>

מנגנוני סyncron: משתני תנאי

הציגת הבעה: תור מקבילי



- מבנה נתונים עם שתי פעולות:
- enqueue – הכנסת איבר לתור.
- dequeue – הוצאה איבר מהתור.
- אם התור ריק, הפעולה תיחסם עד שיכנוס איבר חדש.
- יש להגן על התור ע"י מנעולים.
- כי חוטים שונים יכולים להכנס לתור או להוציאו מהתור במקביל.
- בנוסף צריך להבטיח **סדר**.
- חוט שרוצה להוציא איבר יצטרך להמתין לתנאי "התור אינו ריק".

טור מקבילי הוא מבנה נתונים לפתרון בעיית יצרן-צרכן (producer-consumer). בהרצתה ראיינו פתרון לבעה באמצעות סמפורים; כאן נראה פתרון בעזרת משתני תנאי.

ניסוי ראשון לפתרון

```
mutex_t m;
int queue_size = 0;

void enqueue(item x) {
    mutex_lock(&m);
    /* add x to tail */
    queue_size++;
    mutex_unlock(&m);
}

item dequeue() {
    mutex_lock(&m);
    while (queue_size == 0);
    /* remove from head */
    queue_size--;
    mutex_unlock(&m);
}
```

- מה הבעיה במימוש המוצע?
- קיפאון (deadlock).
 1. חוט #1 מנסה להוציא איבר אבל התור עדין ריק ← נתקע בLOOP של while.
 2. חוט #2 מנסה להכניס איבר לתור ← נתקע כי חוט #1 עדין מחזק את המניעול.
- אם חוט #1 היה מנסה לוותר על המניעול ולתפוא אותו לסייעוי, היינו נתקלים בבעיה אחרת, של עילוות: בדיקה חוזרת ונשנית על גודל התור מבזבצת זמן מעבד.

משתנה תנאי (condition variable)

- משתנה תנאי הוא אובייקט סינכרון המאפשר לחוט **לצאת להמתנה** בתור **קטע קרייטי**.
 - כמובן, לפנות את המעבד ולצאת לתור המשתנה.
- המשתנה תבוצע עד **לקיום תנאי כלשהו**.
- המשתנה מאפשרת **לאכוף סדר** בביצוע של החוטים.
- שימוש תכניות נכון במשתני תנאי מחייב להגדיר גם:
 1. **משתנה מצב** – החוט עובר להמתנה או חוזר מהמתנה בהתאם לערכו של המשתנה המצב.
 2. **מנעול mutex** – מבטיח לנו אוטומיות והגנה על הקטע הקרייטי.

סכמה כללית למשתני תנאי

```
cond_t c; // should be initialized  
mutex_t m; // should be initialized  
int state_var = 0;
```

- החוט הממתין לאירוע יקרה ל:

```
while (!condition_holds(state_var))  
    cond_wait(&c, &m);
```

מודיע cond_wait מקבלת
אם את המניעו?

- החוט שמסמן לחוטים הממתינים להמשיר יקרה ל:

```
if (condition_holds(state_var))  
    cond_signal(&c);
```

נענה על כך עוד כמה שאלות.

מדוּעַ מִקְבָּלָת גֶּם אֶת המניעול?

שחרור המניעול

ואז יציאה למתנה?

```
item dequeue () {
    mutex_lock (&m);
    while (queue_size == 0) {
        mutex_unlock (&m);
        cond_wait (&c);
        mutex_lock (&m);
    }
    /* remove from head */
    queue_size--;
    mutex_unlock (&m);
}
```

יציאה למתנה

ואז שחרור המניעול?

```
item dequeue () {
    mutex_lock (&m);
    while (queue_size == 0) {
        cond_wait (&c);
        mutex_unlock (&m);
        mutex_lock (&m);
    }
    /* remove from head */
    queue_size--;
    mutex_unlock (&m);
}
```

שני המימושים שגויים ← מימוש תקין של משתני תנאי ח"ב לשחרר את המניעול וליצאת למתנה באופן אוטומטי.

המימוש הימני שגוי מכיוון שאחרי יצאה לטור המתנה לא נועור לשורה הבאה (כלומר עדין נחזיק את המניעול).

המימוש השמאלי שגוי מכיוון שחרור המניעול החוט השני יכול לרוץ ולשלוח סיגナル שלר לאיבוד.

לכן הממשק של `cond_wait()` שונה מזו שמצווג בשקף.
הfonקציה `(cond_wait())` מקבלת שני פרמטרים – מניעול ומשתנה תנאי – ואז משחררת את המניעול ויצאת למתנה באופן אוטומטי.

מימוש תיקין

```

cond_t c; // should be initialized
mutex_t m; // should be initialized
int queue_size = 0;

void enqueue(item x) {
    mutex_lock(&m);
    /* add x to tail */
    queue_size++;
    cond_signal(&c);
    mutex_unlock(&m);
}

item dequeue() {
    mutex_lock(&m);
    while (queue_size == 0) {
        cond_wait(&c, &m);
    }
    /* remove from head */
    queue_size--;
    mutex_unlock(&m);
}

```

האם ניתן להוציא את signal מחוץ לנעילה?

שאלת: האם ניתן להוציא את signal מחוץ לנעילה?
 תשובה: כן, ניתן לקרוא `-()signal` גם מחוץ לקטע הקרייטי והקוד עדין יעבד כשרורה.
 אבל מבחינת ביצועים עדיף לקרוא `-()signal` בתוך הקטע הקרייטי, כי אין טעם לשחרר את המנעול לפני שהערכנו את החוט השני שימתין למניעול גם כן.

אתחול ופינוי משתני תנאי

```
#include <pthread.h>
```

```
int pthread_cond_init(pthread_cond_t *cond,  
                      pthread_condattr_t *cond_attr);
```

- ערך מוחזר: הפעולה תמיד מצליחה ומוחזירה 0.

```
int pthread_cond_destroy(pthread_cond_t *cond);
```

- ערך מוחזר: 0 בהצלחה, ערך שונה מ-0 בכישלון (למשל, אם יש עדין חוטים המסתווים על משתנה התנאי).

- פרמטרים:

- cond – משתנה התנאי עליו מבוצעת הפעולה.
- cond_attr – מגדר את תכונות המשתנה התנאי.
- תמיד נועביר ערך NULL בקורס זה.

המתנה על משתני תנאי

```
int pthread_cond_wait(pthread_cond_t *cond,  
                      pthread_mutex_t *mutex);
```

פעולה:

1. משחררת את המניעול ומעבירה את החוט להמתין על משתנה התנאי **באופן אטומי** (ראינו קודם מודיע זה הכרחי).
 - החוט הממתין חייב להחזיק במניעול mutex לפני הקרייה.
 2. בחזרה מהמתנה על משתנה התנאי, החוט **עובר להמתין על המניעול**. החוט יחזיר מהקרייה ל-**(-)pthread_cond_wait** רק לאחר שינעל מחדש את mutex.
- **ערך מוחזר:** הפעולה תמיד מצלילה ומוחזירה 0.

זהו מימוש לפי סמנטייקת Mesa. בהמשך נראה גם את סמנטייקת Hoare.

שחרור חוטים ממתיינים

- ```
int pthread_cond_signal(pthread_cond_t *cond);
```
- משחררת את **אחד** החוטים הממתינים (הגינות לא מובטחת).
- ```
int pthread_cond_broadcast(pthread_cond_t *cond);
```
- משחררת את **כל** החוטים הממתינים.
 - כל החוטים מפסיקים להמתין על משתנה התנאי וועברים להמתין על המneauל. החוטים יჩזרו לפעולות בזיה אחר זה (בסדר כלשהו, לאו דווקא הוגן) לאחר שיגעלו מחדש את mutex.
- **שים לב:** אם אין אף חוט שממתין באותו רגע על משתנה התנאי cond, **הפעולות חסרות השפעה** (הסיגナル הולך לאיבוד ואין נזכר הלאה).
 - **ערך מוחזר:** הפונקציות תמיד מצלוחות ומחזירות 0.

מימוש שגוי #1

הסבירו מדוע הקוד שגוי, כולם תארו בטמיון ורוחיש מסויים שבו הקוד לא פועל כנדרש.

אם החוט הראשון יתבצע לפני השני, האיתות י לר איבוד והחוט השני ית��ע לנצח.

```
cond_t c; // should be initialized
mutex_t m; // should be initialized

void enqueue(item x) {
    mutex_lock(&m);
    /* add x to tail */
    cond_signal(&c);
    mutex_unlock(&m);
}

item dequeue() {
    mutex_lock(&m);
    cond_wait(&c, &m);
    /* remove from head */
    mutex_unlock(&m);
}
```

הבעיה במימוש זהה היא שלא משתמשים במשתנה מצב (כמו `queue_size`) במימוש התקין.

מימוש שагי #2

הסבירו מדוע הקוד שגוי, ככלומר תארו בטמיון ורוחיש מסויים שבו הקוד לא יפעיל כנדרש.

אם תתרחש החלפת הקשר בנקודה ה^ז, האיותות שוב יירלאיבוד.

```
cond_t c; // should be initialized
mutex_t m; // should be initialized
int queue_size = 0;

void enqueue(item x) {
    mutex_lock(&m);
    /* add x to tail */
    mutex_unlock(&m);
    cond_signal(&c);
    queue_size++;
}

item dequeue() {
    mutex_lock(&m);
    while (queue_size == 0) {
        cond_wait(&c, &m);
    }
    /* remove from head */
    queue_size--;
    mutex_unlock(&m);
}
```

הבעיה במימוש זהה היא גישה לא מוגנת למשתנה המשותף `queue_size`. לכן יש לעדכן אותו רק תחת נעליה של ה-`mutex`.

מימוש שגוי #3

מה הבעיה במימוש
זהה?

הפתרון זהה בזבוני
כמו ה-*busy-wait*
שראינו בהתחלה.

```
cond_t c; // should be initialized
mutex_t m; // should be initialized
bool is_signal_caught = false;

void enqueue(item x) {
    mutex_lock(&m);
    /* add x to tail */
    mutex_unlock(&m);
    while (!is_signal_caught) {
        cond_signal(&c);
    }
    is_signal_caught = false;
}

item dequeue() {
    mutex_lock(&m);
    cond_wait(&c, &m);
    /* remove from head */
    mutex_unlock(&m);
    is_signal_caught = true;
}
```

זאת אחת הטעויות הנפוצות של סטודנטים: שליחת הסיגנל בלולאה כדי שלא יLR לאיבוד.

שימוש לב: הפתרון המופיע בשקף שגוי גם כי הוא לא תומך בהכנסה והוצאה רב פעמיים.
תרחיש בעייתי לדוגמה:
 enqueue (until unsetting is_signal_caught) → dequeue (until the end) → dequeue
 . (until cond_wait) → enqueue (until the end)
 החוט האחרון שנכנסה להוצאה ייתקע בהמתנה לנצח.

```

cond_t c;
mutex_t m;
int queue_size = 0;

void enqueue(item x) {
    mutex_lock(&m);
    /* add x to tail */
    queue_size++;
    cond_signal(&c);
    mutex_unlock(&m);
}

item dequeue() {
    mutex_lock(&m);
    if (queue_size == 0)
        cond_wait(&c, &m);
    /* remove from head */
    queue_size--;
    mutex_unlock(&m);
}

```

t2

t1

מימוש שגוי #4

- אם השתמש בתנאי `if` במקום בולולאת `while` יתכן מצב של הוצאת איבר מיותר ריק:
 - .1 בהתחלה התור ריק.
 - .2 חוט t1 קורא ל-`dequeue()` ולכן משחרר את המנעול ומתמיין.
 - .3 חוט t2 קורא ל-`enqueue()`, מכניס איבר לתור ומבצע `cond_signal()`.
- חוט t1 מתעורר וועבר להמתין לשחרור המנעול.

```

cond_t c;
mutex_t m;
int queue_size = 0;

void enqueue(item x) {
    mutex_lock(&m);
    /* add x to tail */
    queue_size++;
    cond_signal(&c);
    mutex_unlock(&m);
}

item dequeue() {
    mutex_lock(&m);
    if (queue_size == 0)
        cond_wait(&c, &m);
    /* remove from head */
    queue_size--;
    mutex_unlock(&m);
}

```

מימוש שגוי #4

- .4 חוט t3 קורא ל-`dequeue()` וnochם בהמתנה למניעול בתחילת הקוד.
- .5 חוט t2 משחרר את המניעול ומוסים את `(enqueue`.
- .6 חוט t3 מקבל את המניעול, נכנס, מוציא איבר ומוסים.
- כלומר חוט t3 משחרר את המניעול.
- .7 חוט t1 מקבל את המניעול, ממשיר לבצע את הקוד ומנסה להוציא איבר מתוך ריק!

מימוש שגוי #4

- ממה נבעה הבעיה?
- בסמנטיקה הנוכחית (Mesa) פועלת (`cond_signal`) לא בהכרח גורמת לחוט הממתין להמשיך מיד, מפני שהחוט צריך לתפוס קודם קודם את המניעול.
- אבל "יתכן" שלפני שהחוט הממתין ינעל את ה-`mutex`, חוט נוסף יירוץ וישנה את הנתונים כך שהמצב הרצוי כבר לא מתקין.

• שאלה: האם עדין הייתה בעיה אם המניעול היה הוגן? (סדר FIFO)

- כיצד ניתן לפתור את הבעיה?
- ע"י בדיקה נוספת של תנאי האירוע לאחר החזרה מ-`cond_wait` והמתנה נוספת לפני היצור. לדוגמה:

```
while (queue_size == 0)
    cond_wait(...)
```

השאלה היא האם הבעיה יכולה לקרות גם אם המניעול היה הוגן (כלומר, אם חוטים תופסים את המניעול לפי סדר הגעתם - FIFO)?

תשובה: כן, עדין הייתה בעיה! אם `t3` היה קורא ל-`dequeue` ומacha למןיעול לפני `t2` הגיעו ל-`signal`.

באופן עקרוני, ה-`while` נדרש תמיד גם בגלגול תופעה של התעוררות שווה מהמתנה (**Spurious Wakeups**).



משתני תנאי בסמנטיקה Hoare

- בשימוש לפי סגנון Hoare, החוט שמבצע (`cond _signal`) גם מותר על המניעול וublisher אותו לחוט הממתין.
- יתרונות: החוט הממתין ממשיר כאשר הוא יודע שתנאי ההמתנה אכן מתקיים.
- חררון: החוט שביצע (`cond _signal`) מותר על ההתקדמות שלו וממתין עד שהמניעול משוחרר, כלומר הוא גענש על כרך שסייע לחוט אחר להתקדם.
- חרון נוסף: סמנטיקת Hoare מסובכת יותר למימוש מסמנטיקת Mesa, ולכן כמעט ולא נמצאת בשימוש.
- שאלת: האם הבעה שראינו בדוגמה התור המקבילי יכולה לקרות אם משתנה התנאי היה בסגנון Hoare?
- תשובה: לא! כי t_1 יהיה מקבל את המניעול ישירות מ- t_2 , לפני שחוט t_2 יוכל להיות לרצוי.

מנגנוני סנכרון: סמפורים

דיקסטרה (מציא הסמפור) היה הולנדי, ובהולנדית סמפור זה "האיש שמכoon את הספינות בים".

סמפור (Semaphore)

- סמפור הוא אמצעי סנכרון אשר מאפשר להבטיח אוטומטיות (כמו מנעול) או סדר (כמו משתנה תנאי) – בהתאם לערך ההתחלתי שלו.
- יכול למשג גם פעולות סנכרון אחרות – נראה בהמשך.

- סמפור ממומש כמוינה אי-שלילי עם שתי פעולות עליון:

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

- פעולה: אם המונה גדול מ-0, מקטינה אותו ב-1.
אחרת, מעבירה את החוט לתור המתנה.

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

- פעולה: אם TOUR הממתינים לא ריק, מוציאה ומעבירה את החוט הראשון בתור. אחרת, מגדילה את המונה ב-1.

הערה: בהרצאה סמפור מוגדר אחרת, כאשר המונה יכול לקטן גם מתחת ל-0.
כמו כן, פעולות post נקראות signal().



סמפור (Semaphore)

- פעולות נוספות על סמפורים:

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

- גרסה לא-חווסמת של wait. אם המונה של הסמפור אינו גדול מ-0, חוזרת מיד ונכשלת.

```
int sem_getvalue(sem_t *sem, int *sval);
```

- קוראת את ערך מונה הסמפור למקום אליו מצביע sval.
 - תמיד מצביעה ומחזירה 0.

אתחול ופינוי סמפור

- יש לכלול קובץ header נוסף מעבר ל- `pthread.h`

```
#include <semaphore.h>
```

- וכמוון לחבר למספריה `pthread` עם דגל הקומpileציה `-pthread`.

- אתחול סמפור לפני השימוש:

```
int sem_init(sem_t *sem, int pshared,  
              unsigned int value);
```

- פרמטרים:

- `sem` – הסמפור עליו מבוצעות הפעולות.

- `pshared` – אם ערכו גדול מ-0, מצביע שהסמפור יכול להיות משותף למספר תהליכים. תוכנה זו אינה נתמכת, ולכן תמיד נציב בו 0.

- `value` – ערךו ההתחלתי של מונחה הסמפור.

- ערך מוחזר: 0 בהצלחה, (-1) בכישלון.

- פינוי סמפור בהתאם השימוש:

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

דוגמה: סמפור בתור מנעול

```
sem_t sem;  
sem_init(&sem, 0, 1);  
sem_wait(&sem);  
// critical section  
sem_post(&sem);
```

- סמפור עם ערך התחלתי 1 נקרא סמפור בינהרִי.
- סמפור בינהרִי יכול לשמש להגנה על קטע קרייטי (ע"י מניעה הדדית בין החוטים הניגשים).

- **dagsh:** סמפור בינהרִי שונה מהמנעול mutex, משומש שכלי חוט יכול לבצע post על סמפור, גם אם לא ביצוע wait על הסמפור קודם לכך (אין "בעלות" על הסמפור).

דוגמה: סמפור בתור מניעול "משוכלל"

- סמפור יכול למשה הגנה על קטע קרייטי מפני הרצה של יותר מ- N חוטים במקביל, אם נתা�חל אותו לערך $1 > N$.

```
sem_t sem;  
  
sem_init(&sem, 0, 10);  
  
void login() {  
    sem_wait(&sem);  
}  
  
void logout() {  
    sem_post(&sem);  
}
```

- דוגמה: שרת שיכל לשרת עד 10 משתמשים.
- במקרה ויהיו 10 משתמשים במערכת, המשתמשים הבאים שיינסו להתחבר יחכו בפקודה `wait()`.
- רק לאחר שימוש כלשהו יתנתק, יורשה להיכנס המשתמש הבא.

דוגמה: סמפור להבטחת סדר

אם נאתחל את הסמפור ל-0, החוט השני ייכה לראשו.

השימוש בסמפור פשוט יותר מאשר במשתנה תנאי כי post() של סמפור לא הולך לאיבוד.

```
mutex_t m;
sem_t queue_size;
sem_init(&queue_size, 0, 0);

void enqueue(item x) {
    mutex_lock(&m);
    /* add x to tail */
    mutex_unlock(&m);
    sem_post(&queue_size);
}

item dequeue() {
    sem_wait(&queue_size);
    mutex_lock(&m);
    /* remove from head */
    mutex_unlock(&m);
}
```

הפוך



דוגמה: מימוש מנעל קוראים-כותבים

מנעול קוראים-כותבים

- מנגנון סנכרון המאפשר להגן על מבנה נתוניים באופן הבא:
 - מספר חוטים יכולים לקרוא את המידע (ambil לשנות אותו) בו-זמנית.
 - כאשר חוט רוצה לעדכן את המידע, הוא צריך גישה בלעדית למבנה הנתוניים.
- לדוגמה, כדי להגן על משתנה x הנגיש ממספר חוטים:

reader thread	writer thread
read_lock(); $y = 2*x;$ read_unlock();	write_lock(); $x = 5*x + 1;$ write_unlock();

- בש侃פים הבאים נדגים כיצד ניתן למשתמש במנעול קוראים-כותבים באמצעות משתני תנאי.
 - בהרצתה ראייתם איך להשתמש בסמפור למטרה זו.

1

שאלה מבחן

מנעול קוראים-כותבים

- משמעותו של מנגנון קוראים-כותבים בעצרת משתני תנאי ומנעולי mutex בלבד (בשונה מהIMPLEMENTATION שראיתם בהרצאה באמצעות סמלים סטנדרטיים).
- יש למשוך את 5 הפונקציות הבאות:
 - reader_lock() .1
 - reader_unlock() .2
 - writer_lock() .3
 - writer_unlock() .4
 - readers_writers_init() .5

מימוש מנעול קוראים-כותבים (1)

```
int readers_inside, writers_inside;
cond_t read_allowed;
cond_t write_allowed;
mutex_t global_lock;

void readers_writers_init() {
    readers_inside = 0;
    writers_inside = 0;
    cond_init(&read_allowed, NULL);
    cond_init(&write_allowed, NULL);
    mutex_init(&global_lock, NULL);
}
```

מה ערכו המקסימלי של
writers_inside

מימוש מנעול קוראים-כותבים (2)

```
void reader_lock() {
    mutex_lock(&global_lock);
    while (writers_inside > 0)
        cond_wait(&read_allowed, &global_lock);
    readers_inside++;
    mutex_unlock(&global_lock);
}

void reader_unlock() {
    mutex_lock(&global_lock);
    readers_inside--;
    if (readers_inside == 0)
        cond_signal(&write_allowed);
    mutex_unlock(&global_lock);
}
```

למה משתמשים בלולאת
while ולא תנאי if?

תשובה: צריך לבדוק את התנאי גם לאחר wait() כפ"י שראינו קודם.

מימוש מנעול קוראים-כותבים (3)

```

void writer_lock() {
    mutex_lock(&global_lock);
    while (writers_inside + readers_inside > 0)
        cond_wait(&write_allowed, &global_lock);
    writers_inside++;
    mutex_unlock(&global_lock);
}

void writer_unlock() {
    mutex_lock(&global_lock);
    writers_inside--;
    if (writers_inside == 0) {
        cond_broadcast(&read_allowed);
        cond_signal(&write_allowed);
    }
    mutex_unlock(&global_lock);
}

```

האם יש צורך ב-if?

למה לא להשתמש ב-broadcast כדי להעיר את כל הכותבים?

תשובה 1#: לא, אין צורך ב-if כי התנאי בהכרח מתקיים.
 תשובה 2#: כי לפיה ההגדרה רק כותב אחד יכול להיות בקטע הקרייטי. אם נuir את כל הכותבים, כל היותר כותב אחד יתקדם והשאר יחזרו להמתין.

חסרוןות של המימוש

- **הרעות כתובים וחסור הוגנות:** כל עוד המנעול אצל הקוראים, קורא חדש שmag יצליח להיכנס ויעקוף כתובים שהגיעו לפניו.
- **חסור סדר:** לא ניתן לדעת האם הקוראים או הכותב יכנסו ל鎖 עקריטי.
 - תלוי מי יצליח לתפוס ראשון את המנעול `global lock` שמשתחרר בסיום `writer_unlock()`.
- איך אפשר לפתור בעיות אלו?

https://en.wikipedia.org/wiki/Readers%20%93writers_problem

2

שאלה מבחן

מועד א', אביב 2008, שאלה 1

- נרצה למשר מנעול קוראים/כותבים עם עדיפות לכותבים.
- בעדיפות לכותבים הכוונה שם יש גם קוראים וגם כותבים המוחכים להיכנס לקטע הקרייטי, הכותבים מקבלים עדיפות – יכנסו **תמיד** לפני הקוראים.
- **סעיף א:** סמןנו בעיגול את כל הדרישות מפרטן הבעה החדשה.

- | | |
|--|--|
| יכול להיות לכל היותר קורא אחד בקטע קרייטי.
יכול להיות לכל היותר כתוב אחד בקטע קרייטי.
יכולים להיות מספר קוראים בקטע קרייטי.
יכולים להיות מספר כותבים בקטע קרייטי.
אסור לכותבים וקוראים להיות בקטע קרייטי בו זמן.
אסור להריעיב קוראים שמנסים להיכנס לקטע קרייטי.
אסור להריעיב כותבים שמנסים להיכנס לקטע קרייטי.
יתכן מצב שקורא שהגיא אחריו כתוב יכנס לקטע הקרייטי לפניו.
יתכן מצב שכותב שהגיא אחריו קורא יכנס לקטע הקרייטי לפניו. | X
V
V
X
V
X
V
X
V |
|--|--|

הערה לגבי שתי הנקודות האחרונות: הניסוח לא מספיק ברור, כי המינוח "הגיא אחריו" לא מוגדר חד-משמעות.

הניסוח המדויק הוא כפי שהוגדר בתחלת השאלה, כלומר אם יש גם קוראים ממתיינים וגם כותבים ממתיינים, אז הכותבים מקבלים עדיפות.

לדוגמא: נניח כי כרגע יש כתוב ייחיד שסימן את ()`write_lock`, אבל עוד לא התחיל את ()`write_unlock`.

כעת הגיא קורא ויצא להמתנה ב-()`read_lock`. לאחר מכן הגיא כתוב נוסף וגם יצא להמתנה ב-()`write_lock`.

לאחר שהכותב הראשון יעזוב, נרצה שהכותב השני יכנס למראות שהוא הגיא אחריו הקורא.

מועד א', אביב 2008, שאלה 1

```
sem_t sem; // Global semaphore,  
with initial value 1  
  
int writer_lock() {  
    sem_wait(sem);  
}  
  
int writer_unlock() {  
    sem_post(sem);  
}  
  
int reader_lock() {  
    while(sem_getvalue(sem) <= 0)  
        sleep(1);  
    sem_wait(sem);  
}  
  
int reader_unlock() {  
    sem_post(sem);  
}
```

- סעיף ב: להלן הצעה לפתרון בעית קוראים/כותבים עם עדיפות לכותבים, המשמשת בסמפורים.

- תארו 3 בעיות שונות של נכונות /או יעילות שיש בפתרון הנ"ל. הוכיחו כי הסמפור הינו הוגן.

מועד א', אביב 2008, שאלה 1

```
sem_t sem; // Global semaphore,  
with initial value 1  
  
int writer_lock() {  
    sem_wait(sem);  
}  
  
int writer_unlock() {  
    sem_post(sem);  
}  
  
int reader_lock() {  
    while(sem_getvalue(sem) <= 0)  
        sleep(1);  
    sem_wait(sem);  
}  
  
int reader_unlock() {  
    sem_post(sem);  
}
```

1. **בעיית נכונות:** הפתרון לא מאפשר ליותר מקרוא אחד להיכנס לקטע קרייטי.
2. **בעיתת נכונות:** אם יש גם קוראים וגם כתובים, הכותבים לא בהכרח יקבלו עדיפות ועלולים להיות מורעבים בняgod לדרישה.
3. **בעיתת ייעילות:** קוראים מבצעים wait busy.

מועד א', אביב 2008, שאלה 1

- **סעיף ג:** כתבו קוד הפותר את בעית קוראים/כותבים עם עדיפות לכותבים, המשמש במנועלים ומשתני תנאי.
- ניתן להציג משתנים גלובליים ומציע סנסרן כרצונכם, (מנועלים, ומשתני תנאי) אבל יש לזכור כי יעילות הפתרון מהוות חלק מהצין (כלומר מיעוט מציע הסנסרן עדיף וקטועים קרייטיים קצרים עדיפים). ניתן להניח שעדיות כל החוטים זהה ומציע הסנסרן הינם הוגנים.
- **רמז:** מומלץ להיעזר בפתרון הבעיה של מנועל קוראים/כותבים עם עדיפות לקוראים, כפי שהוצגה בתרגול.

מימוש מנעול קוראים-כותבים (1)

```
int readers_inside, writers_inside, writers_waiting;
cond_t read_allowed;
cond_t write_allowed;
mutex_t global_lock;

void readers_writers_init() {
    readers_inside = 0;
    writers_inside = 0;
    writers_waiting = 0;
    cond_init(&read_allowed, NULL);
    cond_init(&write_allowed, NULL);
    mutex_init(&global_lock, NULL);
}
```

מועד א', אביב 2008, שאלה 1

```
void reader_lock() {
    mutex_lock(&global_lock);
    while (writers_inside > 0 || writers_waiting > 0)
        cond_wait(&read_allowed, &global_lock);
    readers_inside++;
    mutex_unlock(&global_lock);
}

void reader_unlock() {
    mutex_lock(&global_lock);
    readers_inside--;
    if (readers_inside == 0)
        cond_signal(&write_allowed);
    mutex_unlock(&global_lock);
}
```

מועד א', אביב 2008, שאלה 1

```
void writer_lock() {
    mutex_lock(&global_lock);
    writers_waiting++;
    while (writers_inside + readers_inside > 0)
        cond_wait(&write_allowed, &global_lock);
    writers_waiting--;
    writers_inside++;
    mutex_unlock(&global_lock);
}

void writer_unlock() {
    mutex_lock(&global_lock);
    writers_inside--;
    if (writers_inside == 0) {
        cond_broadcast(&read_allowed);
        cond_signal(&write_allowed);
    }
    mutex_unlock(&global_lock);
}
```

סינכרון בגרעין לינוקס

alogiyah ha-mesuda

- דמיינו מסעדה ובה המלצר מטפל בשני סוגים של לקוחות:

לקוחות VIP	לקוחות רגילים
המלצר מטפל מיד בכל לקוח VIP שמגיע, גם אם צריך לעזוב באמצעות לקוח אחר (רגיל או VIP).	אם המלצר פניו ומגיע ללקוח רגיל, אז המלצר עובד לטפל בו.
לקוח VIP לעולם לא ישחרר את המלצר שמטפל בו כרגע.	לקוח רגיל יכול לשחרר את המלצר שמטפל בו כרגע לטובת לקוח אחר.
המלצר לא עוזב ללקוח VIP לטובת לקוח רגיל.	המלצר יכול לעזוב ללקוח רגיל לטובת לקוחות VIP שmagim. לאחר הטיפול בלקוחות VIP, המלצר יכול לחזור לטפל בלקוח רגיל אחר.

מלצר יחיד יכול לטפל במספר לקוחות בו-זמנית, למשל: להתחילה לטפל בלקוח רגיל, לעבור ללקוח VIP שהגיע פתאום, ולאחר מכן לחזור ללקוח הרגיל.
כאשר יש כמה מלצרים, כל אחד מהם יכול לטפל בלקוח אחר.

הגרעין הוא מלצר

- הגרעין מטפל בשני סוגים בקשנות (לקוחות):

פסיקות חומרה	קריאות מערכת / חריגות
הגרעין מטפל מיד בכל פסיקת חומרה שמגיעה, גם אם הוא בא מצב טיפול בחrigה / קריית מערכת / פסיקת חומרה אחרת.	אם המעבד מריץ קוד משתמש ומגיעה קריית מערכת / חריגה אז הגרעין עובר לטפל בה.
שגרת טיפול בפסיקת חומרה לעולם לא תועור על המעבד.	קריאות מערכת יכולות ליותר על המעבד, לדוגמה <code>(() read</code> .
שגרת טיפול בפסיקת חומרה לעולם לא תקרה לקריית מערכת או ליצור חריגה (למעט חריגת דף - page fault).	הגרעין יכול לקטוע טיפול בקריאת מערכת / חריגה לטובת פסיקות חומרה שмагיעות. לאחר הטיפול בפסיקות החומרה, הגרעין יכול לעבור לטפל בתהילך אחר מזה שרך קודם.

בהקבלה למשל המסעדה: מלצר פנוי == המעבד נמצא במצב משתמש. מעבד יחיד יכול לטפל במספר פסיקות "בו-זמןית", למשל: להתחילה לטפל בקריאת מערכת, ואז לעבור לטפל בפסיקת שעון שהגיעה פתאום. במערכת עם מספר מעבדים, כל מעבד יכול לטפל בפסיקה שונה.

איך זה קשור לבעיות סנכרון?

- נסתכל על התרחיש הביעיתי הבא:
 - במסעדת יש ערכות תה אחת המורכבות ממספר חלקים (קנקן, כוסות, ...).
 - לקוחות רגילים נכנסים למסעדה ומבקשים תה.
 - המלצר מתחילה לעבוד ומגיש ללקוח את הקנקן.
 - לפטוע נכנס לקוח VIP וגם מבקש תה. המלצר מבונן ניגש לשרתת אותו מיד.
 - המלצר מעביר לו את הכוסות, אבל הקנקן עדין אצל הלוקו הקודם.
 - כל לקוח מחזיק חלק מהערכה בגלל שהמלצר לא הביא אותה **בצורה אוטומטית**.
- ההקללה לגרעין המשרת פסיקות: **בעיית אוטומיות בגישה למשתנים מסווגים**.

מסלול בקירה בגרעין

- **מסלול בקירה בגרעין** (kernel control path) הוא רצף פקודות שהגרעין מבצע כדי לטפל ב:
 1. **קריאה מערכת** – בקשה שירות מצד תהליך משתמש.
 - למשל `(fork()` או `(getpid()`).
 2. **פוסיקת תוכנה (חריגה)** – שגיאה שנוצרת ע"י קוד משתמש.
 - למשל חלוקה באפס.
 3. **פוסיקת חומרה** – פסיקה אסינכרונית מהתקן חומרה חיצוני.
 - למשל פסיקת שעון.

טעות נפוצה של סטודנטים: קריית מערכת חוסמת אינה חוסמת פסיקות.

מסלול בקירה נחתכים

- מסלולי בקירה עלולים לחזור זה את זה או להשתלב (interleave) זה בזזה. לדוגמה:
 - לפני סיום הביצוע של קריית המערכת (fork), התקבלה פסיקת שעון, אשר גרמה לביצוע של (scheduler_tick) ← פסיקת חומרה חתכה קריית מערכת.
 - תהלייר A קרא לקריית המערכת (wait, יצא להמתנה והעביר את המעבד לתהלייר B. תהלייר B קורא ביןתיים לקריית המערכת (getpid) ← קריית מערכת חתכה קריית מערכת.
 - שני מעבדים שונים מטפלים בו-זמנית בשתי חריגות שיצרו התהלייכים שריצו עליהם ← חריגה חתכה חריגה.
- יש להגן על מבני נתונים בגרעין הנגישים למסלולי בקירה נחתכים.
 - גישה למבנה נתונים של הגרעין מהוות קטע קרייטי שחייב להתבצע בשלמותו ע"י מסלול הבקירה שנכנס אליו לפני שמסלול בקירה אחר יוכל להיכנס אליו.

שימוש לב: החיתוך הוא בין מסלולי קוד בגרעין. אין משמעות לתהלייכים המעורבים.

אילו חיתוכים אפשריים?

- במערכת מעבד יחיד:**

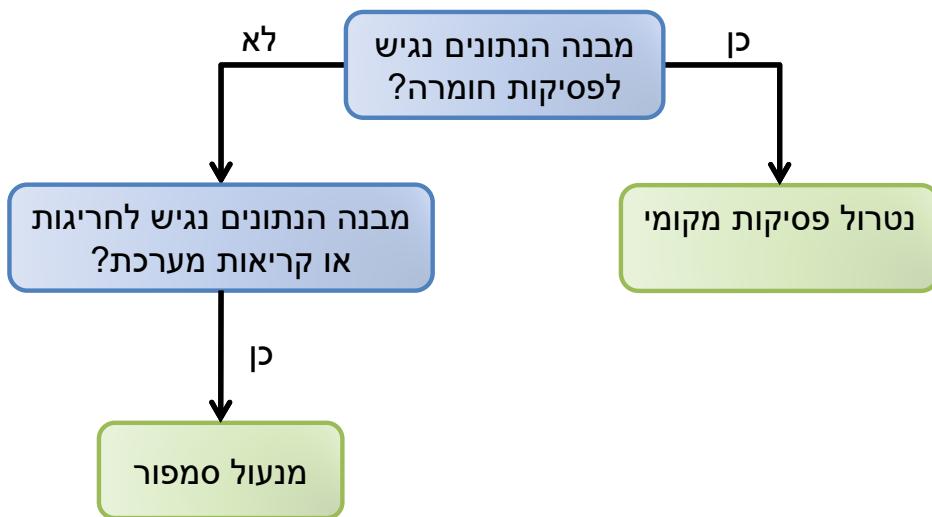
סוג	יכולת להיחתך ע"י
קריאת מערכת או חריגה	כל מסלול בקורסה
פסיקות חומרה בלבד	פסיקות חומרה

- מסלולים שאיןם יכולים להיחתך:
- פסיקת חומרה אף פעם לא קוראת לקריאת מערכת.
- פסיקת חומרה אף פעם לא גורמת לחריגה (למעט חריגת דף).
- קריאת מערכת / חריגה אף פעם לא גורמת לחריגה נוספת (למעט חריגת דף).

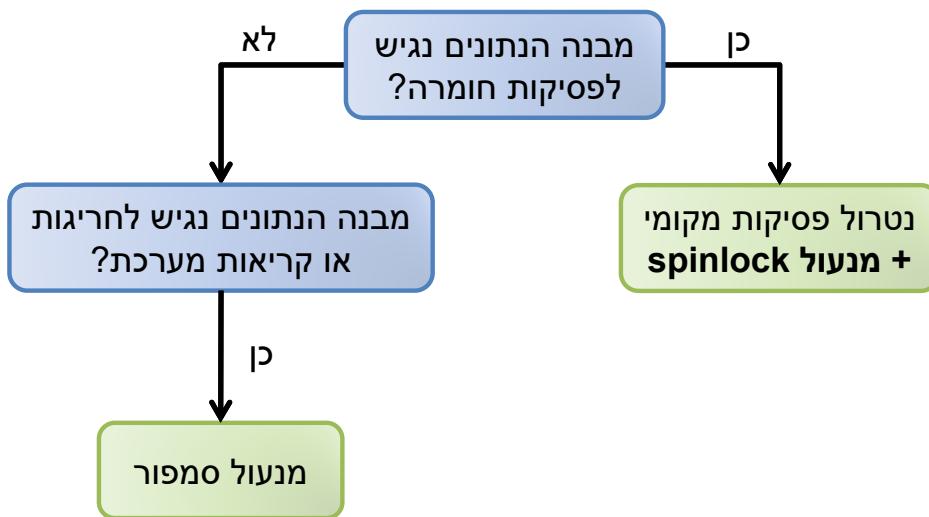
- במערכת מרובת מעבדים:** כל מסלולי הקורסה יכולים להיחתך כי מעבדים שונים יכולים להריץ בו-זמנית מסלולי בקורסה שונים.

שימוש לב: קריאת מערכת/חריגה יכולה להיחתך ע"י קריאת מערכת/חריגה אחרת בגלל שהגרעין ניתן להפקעה (preemptible kernel).

אמצעי הגנה במערכת מעבד יחיד



אמצעי הגנה במערכת מרובת מעבדים



פסיקות חומרה במערכת מעבד יחיד

- פסיקת חומרה חדשה יכולה להגיע לטור-כדי ביצוע טיפול בפסקית חומרה אחרת (קינון פסיקות חומרה).
- פסיקת חומרה יכולה להגיע גם טור כדי טיפול בחריגה.
- כאמור, יש להגן על מבני נתונים הנגישים לפסיקות חומרה באמצעות מנעולים. בפועל, נעה היא בעייתי במערכת עם **מעבד יחיד**. נציגים באמצעות המרחיש הבא:
 - מסלול בקירה #1 מטפל בפסקית חומרה כלשהי ומחזיק מנעול.
 - פסיקת חומרה אחרת מגיעה ומתופלת מיד במסלול בקירה #2 (אשר קוטע את מסלול בקירה #1).
 - מסלול בקירה #2 מנסה לתפוס את המנעול, וכך הוא ממתין לסיום מסלול #1.
 - אבל גם מסלול בקירה #1 ממתין לסיום מסלול #2 לפני שיחזור לרוץ.
 - **קיבלו deadlock**.

נטרול פסיקות מקומי

- כאמור, תפיסת מנעול במהלך טיפול בפסיקות חומרה עלולה להוביל ל-deadlock במערכת עם מעבד יחיד.
- לכן יש להשתמש באמצעי סנכרון אחר: **נטרול פסיקות מקומי** (Local Interrupt Disabling).
- כדי לנטרול פסיקות יש לכבות את הדגל IF של רגיסטר RFLAGS.
- כל עוד $IF == 0$, המעבד המקומי (שמריץ את הקטע הקרייטי) לא יקבל פסיקות חומרה וכך הקטע הקרייטי יתבצע בצורה אוטומטית.
- שימוש לב:** נטרול הפסיקות לזמן רב עלול לגרום לפגיעה בביצועים ולאובדן פסיקות חיוניות, ולכן משתמשים באמצעי זה כמוצא אחרון.

נטרול פסיקות מקומית מתבצעת בצורה הבאה: לפני הקטע הקרייטי שומרים את הדגל IF ואז מצבים לו 0.

בסוף הקטע הקרייטי משחזרים את הערך שנשמר.

למה לא פשוט מכבים את IF לפני הקטע הקרייטי ומדליקים אותו לאחר הקטע הקרייטי?
תשובה: מפני שהדגל IF לא דלק בהכרח לפני הcycles. חסימת פסיקות מתבצעת בכניסה לקטע קרייטי
ויתכן שקטעים קרייטיים שונים מוקוונים זה בזה.

לדוגמה, אם שגרת הטיפול בפסיקת חומרה צריכה לתפוס שני מניעולים, היא תקרא ל:

```
spin_lock_irq(lock1);
...
spin_lock_irq(lock2);
... // starting nested critical section
... // leaving nested critical section
spin_unlock_irq(lock2);
...
spin_unlock_irq(lock1);
```

הגעילה הראשונה תחסום פסיקות, הנעילה השנייה כבר לא משפיעה על הדגל IF. בסיום הקטע
הקרייטי הפנימי, אסור להדליק את הדגל IF כי אנחנו עדין בתוך קטע קרייטי שנעלו אותו.

"When the kernel enters a critical section, it disables interrupts by clearing the IF flag of the eflags register. But at the end of the critical section, often the kernel can't simply set the flag again. **Interrupts can execute in nested fashion, so the kernel does not necessarily know what the IF flag was before the current control path executed.** In these cases, the control path must save the old setting of the flag and restore that setting at the end."

ন্টרול פסיקות מקומי בפונקציה (schedule)

- ישנו מבני נתונים בגרעין הנגישים גם לפסיקות חומרה וגם לקריאות מערכת, למשל תור הריצה (runqueue).
 - קראת המערכת (wait) יכולה להוציא את התהילר הנוכחי מטור הריצה בפונקציה (schedule).
 - פסיקת שעון יכולה להעביר את התהילר הנוכחי למקום אחר בתור הריצה בשגרה (scheduler_tick).
 - אם מסלולי הקריאה של קראת המערכת (wait) ופסיקת השעון "יחתכו", תור הריצה עלול להגיע במצב לא תקין.
-
- **במערכת מעבד יחיד:** נטרול פסיקות מקומי בפונקציה (schedule) הכרחי ומספיק כדי למנוע חיתוך בין מסלולי בקירה כמו בדוגמה הנ"ל.
 - **במערכת מרובת מעבדים:** יש להוסיף מנעול lock/unlock (ראו בשקף הבא...)

פסיקות חומרה במערכת מרובת ליבות

- במערכת מרובת ליבות, מעבדים שונים יכולים לגשת בו-זמןית למבנה נתונים משותפים ↪ יש להוסיף נעליה מעבר לחסימת הפסיקות המקומיות.
- שגרות טיפול בפסיקות חומרה עשוות שימוש במנעולי spinlock (מנעולים המוממשים כת-wait busy).
- שאלה:** מדוע מעדיפים מנעולי spinlock על-פני מנעולי סטפור?
 1. wait busy הוא המתנה עיליה יותר כאשר מדובר בנסיבות קצורות מאוד כי שקרה בגרעין, מפני שכך נחsettת התקורה של כניסה ויציאה מהמתנה.
 2. בעית הוגנות, למשל בתרחיש הבא:
 - תהליך רץ, ובאותו הזמן מתקבלת פסיקת חומרה (למשל מהמקלדת).
 - הטיפול בפסיקה מנסה ל特派 אט המניעל, אבל המניעל כבר תפוץ.
 - התהליך עבר לתור המתנה מסיבה שנייה תלולה בו.

מודדר ע"י טיפוא t_spinlock בקובץ גרעין include/linux/spinlock.h

קריאות מערכת + חריגות

- **כעת נניח** שמבנה נתונים כלשהו **נגיש** לחריגות וקריאות מערכת בלבד (כלומר אינו **נגיש** לפסיקות חומרה).
- מה תרחש **הסינכרון** הביעית'ים?
- **במערכת עם מעבד יחיד:** קרייאות מערכת וחריגות לא מתבצעות בצורה אוטומטית (ביחס לקרייאות מערכת וחריגות אחרות) בגלל שגרעין לינוקס נתן להפקעה.
- **במערכת מרובת מעבדים:** כל קרייאות המערכת והחריגות יכולות להתבצע במקביל על מעבדים שונים.

קריאות מערכת + חריגות

- טקטייקות ההגנה האפשריות:

1. להשאיר את מבנה הנתונים במצב תקין לפני הוייתור על המעבד בקריאה ל-`schedule` – בדרך כלל בלתי אפשרי.
2. חסרון נוסף: בחזרה לביצוע יש לבדוק שהנתונים לא שונו ע"י מסלול אחר.
 - להבטיח אוטומיות בגין המבנה הנתונים ע"י **נעלית סטטוס**.
 - למשל: `(read` מחזיקה מנגנון לכל קובץ שעליו היא עובדת).
 - תהיליך שני שינסה לתפוא את הסטטוס בקריאה `read` יעבור לתוך המתנה. בעtid, התהיליך הראשון ישחרר את הסטטוס והטהיליך השני יתעורר וימשיך בפעולתו.
 - הסטטוס מספק הגנה מפני ביצוע במקביל גם במערכת מרובת מעבדים.

From UTLK2, pages 213–214:

5.5.5 Inode Semaphore

As we shall see in Chapter 12, Linux stores the information on a disk file in a memory object called an *inode*. The corresponding data structure includes its own semaphore in the *i_sem* field.

A huge number of race conditions can occur during filesystem handling. Indeed, each file on disk is a resource held in common for all users, since all processes may (potentially) access the file content, change its name or location, destroy or duplicate it, and so on. For example, let's suppose that a process lists the files contained in some directory. Each disk operation is potentially blocking, and therefore even in uniprocessor systems, other processes could access the same directory and modify its content while the first process is in the middle of the listing operation. Or, again, two different processes could modify the same directory at the same time. All these race conditions are avoided by protecting the directory file with the inode semaphore.

תרגול 9

תקשורת לא אמינה

תקשורת אמינה

תכנות מונחה אירועים

TL;DR

- בשנת 2020, 60% מאוכלוסיית העולם (4.57 מיליארד איש) השתמשו באינטרנט.
- האינטרנט = רשת של רשתות תקשורת בין מחשבים.
- מערכת הפעלה תומכת בתקשורת בין מחשבים על-גבי רשת האינטרנט באמצעות תיווך בין היבומים לחומרה – כרטיס הרשת.
- לינוקס מציג ממשק תכונות מבוסס sockets למימוש תקשורת במודל שרת-לקוח.
- מנגנון ה-sockets מאפשר לתקשר במגוון פרוטוקולים, הנפוצים שבهم הם הפרוטוקולים של רשת האינטרנט:
 - UDP/IP – העברת הודעות בתקשורת לא אמינה.
 - TCP/IP – העברת זרם נתונים בתים בתקשורת אמינה.

מבוא Zariz לתקשרות

מודל השכבות בתקשורת

- נהוג לחלק מערכות תקשורת לשכבות (layers) כאשר לכל שכבה תפקיד מוגדר.

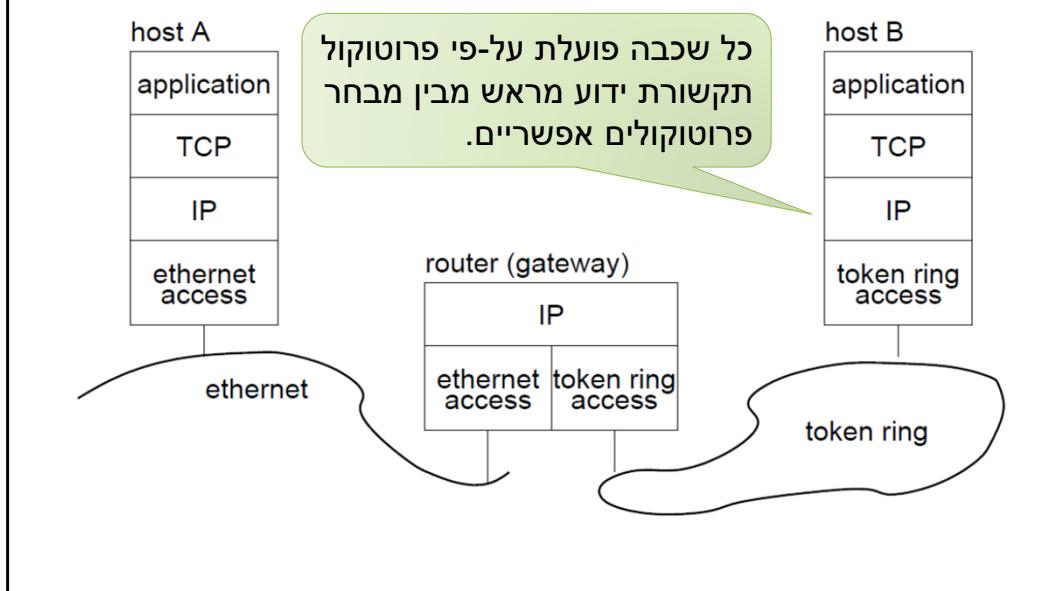
- **בצד השולח:**

- כל שכבה מטפלת בנוטונים (בדרך כלל מוסיף header ל-payload) של השכבה מעלה,
- ועבירה לשכבה מתחתיה.

- **בצד המתקבל:**

- כל שכבה מטפלת בנוטונים (בדרך כלל מסירה את header) של השכבה מתחתיה,
- ועבירה לשכבה מעלה.

מודל השכבות בתקשורת אינטרנט



פרוטוקולי תקשורת באינטראנט

layer	name	protocols	data unit	addressing
5	application	HTTP, SMTP	stream, messages	N/A
4	transport	TCP, UDP	segments, datagrams	ports
3	network	IP	packets	IP
2	link	ethernet	frames	MAC
1	physical	802.11	bits	N/A

איפה מערךת הפעלה בכל הספר?

layer	name	protocols	implemented by
5	application	HTTP, SMTP	user-level code
4	transport	TCP, UDP	
3	network	IP	kernel-level code (the operating system)
2	link	ethernet	
1	physical	802.11	hardware (NIC)

הבעיה המרכזית: אובדן נתונים

- העיקרון הבסיסי ביותר בקשרות הוא שנ נתונים ילו לאי-בוד, יושחתו, או פשוט לא יגיעו ליעדם.
- יש לכך מגוון סיבות:
 1. התקנים בראשת (מודמים, ראותרים, כבלים, ...) עלולים להתקלקל.
 2. קפיצות מתח או קירינת רקע עלולות להפוך חלק מהבאים.
 3. והסיבה הנפוצה מכלם: מחסור בזכרון בתקנים בראשת כדי לאגור את כל המידע שmag'ע.

AIR מתמודדים עם אובדן נתונים?

תקשורת אמינה

- האפשרות השנייה היא למשתמש בתקשורת אמינה באמצעות אמצעים שונים שנראה בהמשך.
- הדוגמה בה נטמקד היא פרוטוקול IP/TCP, אשר מאפשר העברת זרם בתים (stream) כך שכל המידע יגיע בשילמותו ובסדר הנכון.
- מתאים לכל שאר היישומים...

תקשורות לא אמינה

- האפשרות הראשונה היא פשוט להתעלם מהבעיה...
- הדוגמה בה נטמקד היא פרוטוקול UDP/IP, אשר מאפשר העברת הודעות (datagrams) ללא הבטחה שהן יגיעו ליעדם.
- מתאים לישומים כמו VoIP, אשר עבורם השהייה נמוכה חשובה יותר מאמינות.

תקשורת לא אמינה

החומר בפרק זה מבוסס על פרק 48 (Distributed Systems) מהספר .OSTEP

פרוטוקול UDP/IP (User Datagram Protocol)

- הפרוטוקול לא מבטיח תקשורת אמינה, כלומר הודעות יכולות ללקת לאיבוד מבלי ידיעת השולח (וכמובן ללא ידיעת המקלט).
- עם זאת, הפרוטוקול כן מספק הגנה בסיסית מפני שגיאות באמצעות checksums, כלומר אם הודעה הגיעה למקלט, אז ההודעה תקינה בסבירות גבוהה.
- UDP משתמש ב-CRC (cyclic redundancy codes).
- זהו פרוטוקול תקשורת ללא חיבור (connectionless): החלפת המידע בין הצדדים מתחילה מיד, ללא צורך בייסוד חיבור.
- הקללה לעולם האמיתי: משלוח חבילה בדואר.

גם דואר ישראל לא מבטיח תקשורת אמינה כי חבילות יכולות ללקת לאיבוד ☹

מודל שרת-לקוח

הלקוח (client)

- מבקש את השירות.
- אקטיבי – פונה לשרת כאשר הוא זקוק למידע/שירות ממנו.
- דוגמה: הדפדפן הוא תוכנת לקוח המבקשת את פרטי חשבון הבנק או בקשות פעולה שונות בחשבון.

השרת (server)

- מספק את השירות.
- פסיבי – מאזין לרשות ומחכה לקבל בקשות של לקוחות.
- שרת יכול לטפל בבקשות של מספר לקוחות בו זמן נתון.
- דוגמה: שרת <http://> של בנק מאפשר למשתמשים לגשת לחשבון הבנק שלהם.



תכנות מובוא sockets

- לינוקס מאפשרת תקשורת במודל שרת-לקוח באמצעות אוסף Berkeley sockets API ונתונים שנקרו ו-
- קריאות מערכת ומבני נתונים שנקרו ו-
- מחשבים. socket (שקע) הוא נקודת קצה לשיליחה וקבלת של נתונים בראשת מחשבים. socket מאופיין ע"י שני שדות:
 - כתובת שקע מקומית = כתובת IP מקומית + מספר פורט.
- אם השקע מחובר לשקע אחר (ע"י קריית המערכת accept(),
از ה-socket מאופיין ע"י שלושה שדות נוספים:
 - כתובת שקע מרוחקת = כתובת IP מרוחקת + מספר פורט.
 - פרוטוקול – אחד מהפרוטוקולים של שכבה התעבורה: TCP, UDP,

דוגמת קוד שרת-לקוח מעל UDP/IP

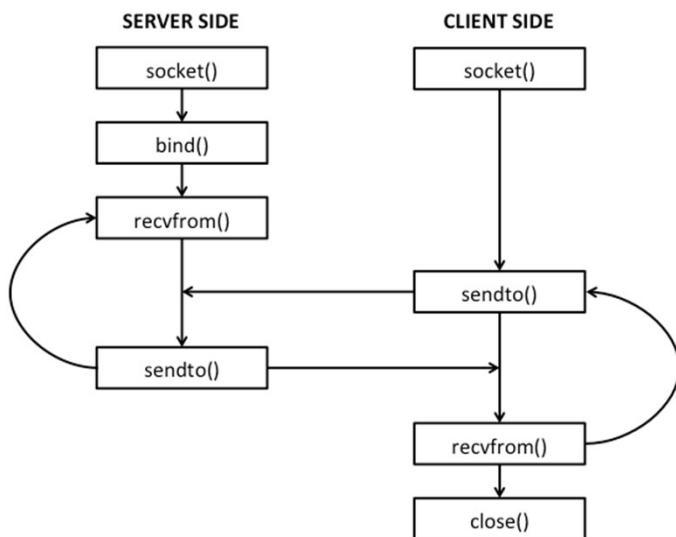


Figure from:

<https://www.it.uu.se/education/course/homepage/dsp/vt19/modules/module-2/sockets/>

client code

```
int main(int argc, char *argv[]) {  
    int sd = UDP_Open(20000);  
    struct sockaddr_in addrSnd, addrRcv;  
    int rc = UDP_FillSockAddr(&addrSnd,  
        "csm.technion.ac.il", 10000);  
    char message[BUFFER_SIZE];  
    sprintf(message, "hello world");  
    rc = UDP_Write(sd, &addrSnd,  
        message, BUFFER_SIZE);  
    if (rc > 0)  
        int rc = UDP_Read(sd, &addrRcv,  
            message, BUFFER_SIZE);  
    return 0;  
}
```

server code

```
int main(int argc, char *argv[]) {  
    int sd = UDP_Open(10000);  
    assert(sd > -1);  
    while (1) {  
        struct sockaddr_in addr;  
        char message[BUFFER_SIZE];  
        int rc = UDP_Read(sd, &addr,  
                           message, BUFFER_SIZE);  
        if (rc > 0) {  
            char reply[BUFFER_SIZE];  
            sprintf(reply, "goodbye world");  
            rc = UDP_Write(sd, &addr, reply, BUFFER_SIZE);  
        }  
    }  
    return 0;  
}
```

UDP_open()

```
int UDP_Open(int port) {
    int sd;
    if ((sd = socket(AF_INET, SOCK_DGRAM, 0)) == -1)
        return -1;
    struct sockaddr_in myaddr;
    bzero(&myaddr, sizeof(myaddr));
    myaddr.sin_family = AF_INET;
    myaddr.sin_port = htons(port);
    myaddr.sin_addr.s_addr = INADDR_ANY; //any ip I have
    if (bind(sd, (struct sockaddr *) &myaddr,
            sizeof(myaddr)) == -1) {
        close(sd);
        return -1;
    }
    return sd;
}
```

UDP_FillSockAddr()

```
int UDP_FillSockAddr(struct sockaddr_in *addr,  
                      char *hostname, int port) {  
    bzero(addr, sizeof(struct sockaddr_in));  
    addr->sin_family = AF_INET; //host byte-order  
    addr->sin_port = htons(port); //network byte-order  
    struct in_addr *in_addr;  
    struct hostent *host_entry =  
        gethostbyname(hostname);  
    if (host_entry == NULL)  
        return -1;  
    in_addr = (struct in_addr *) host_entry->h_addr;  
    addr->sin_addr = *in_addr;  
    return 0;  
}
```

UDP_Write(), UDP_Read()

```
int UDP_Write(int sd, struct sockaddr_in *addr,
char *buffer, int n) {
    int len = sizeof(struct sockaddr_in);
    return sendto(sd, buffer, n, 0,
        (struct sockaddr*)addr, len);
}

int UDP_Read(int sd, struct sockaddr_in *addr,
char *buffer, int n) {
    int len = sizeof(struct sockaddr_in);
    return recvfrom(sd, buffer, n, 0,
        (struct sockaddr*)addr, (socklen_t*) &len);
}
```

נושאים שהتعلמנו מהם

- איך לתרגם שם של שרת (כתובת URL) לכטובות IP?
- הקוד שלנו השתמש בפונקציה (`gethostbyname()`) , אבל החלופה העדיפה `getaddrinfo()`.
 - יותר היא הפונקציה (`htons()`) .
- פרטיהם נוספים – בהרצאה.
- איך להתגבר על סדר בתים שונה במכונות שונות?
 - big endian vs. little endian
- הקוד שלנו השתמש בפונקציה (`htons()`) .
- אם משתמשים בפונקציה (`getaddrinfo()`) אז ניתן להעתים מבעיות כאלה כי הפונקציה (`getaddrinfo()`) מסתירה את הקרייה `htonl()` .

תקשרות אמינה

החומר בפרק זה מבוסס על פרק 48 (Distributed Systems) מהספר .OSTEP.

פרוטוקול TCP/IP (Transmission control protocol)

- ישומם רבים לא יכולים לעבוד מעל תקשורת לא אמינה.
- לא הייתם רוצים שמייל חשוב שלחחים יLR לאיבוד, נכון?
- בדיקן לשם כך הומצא פרוטוקול TCP/IP, אשר מבטיח תקשורת ללא איבוד מידע בדרך. המידע נשלח בצורה זרם (stream) של נתונים ומגיע בבדיקה בסדר בו הוא נשלח.
- זהו פרוטוקול מבוסס-חיבור (connection-oriented): החלפת המידע מתחילה לאחר יסוד חיבור (connection), ועם סיום התקשרות נסגר החיבור.
- הקבלה לעולם האמיתי: שיחת טלפון.

דוגמת קוד שרת-לקוח מעל TCP/IP

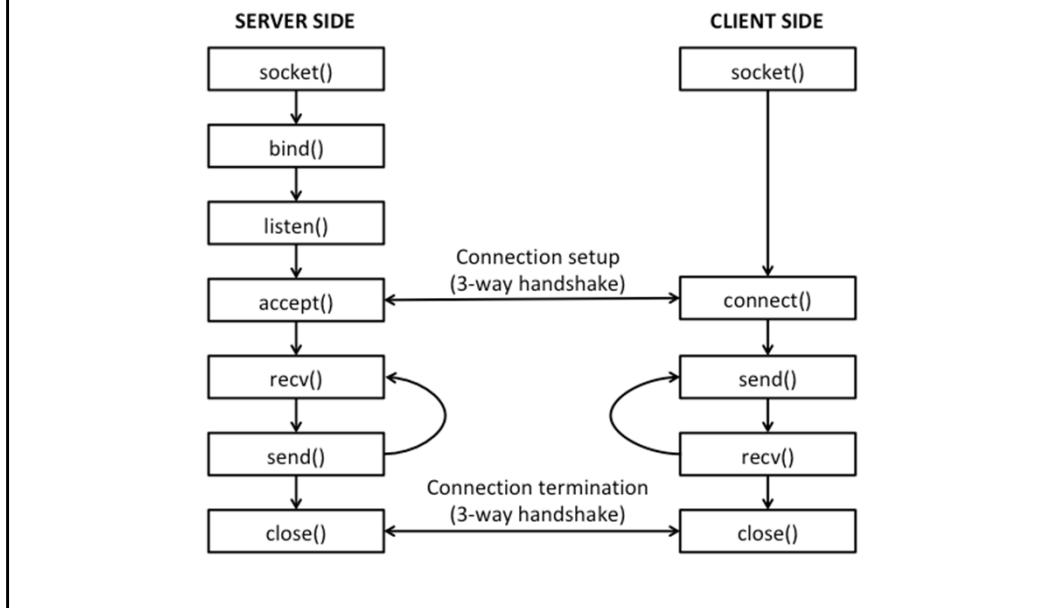
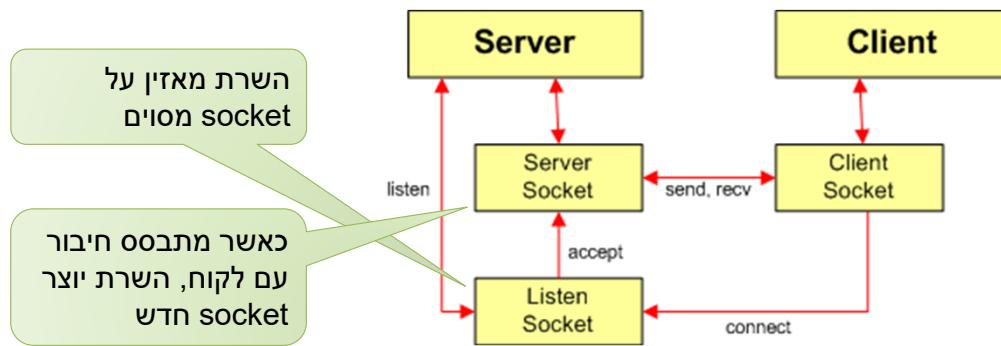


Figure from:

<https://www.it.uu.se/education/course/homepage/dsp/vt19/modules/module-2/sockets/>

דוגמת קוד שרת-לקוח מעל IP/IP/UDP.

- את הקוד המלא ראייתם בהרצאה.
- מבנה הקוד שונה מזה שראינו קודם עבורי IP/IP/UDP.
- השינוי המרכזי הוא יסוד החיבור לפניה תחילת התקשרות.
- באמצעות קריאות המערכת (.bind(), listen(), accept(), connect())

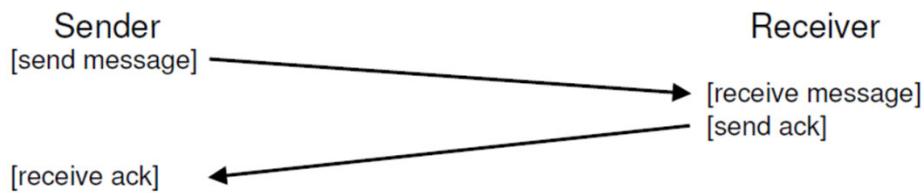


התמונה מתווך:

<https://www.codeproject.com/Articles/9424/Single-Server-Multiple-Clients-Win-MFC-classes-f>

איך מושגים תקשורת אמינה?

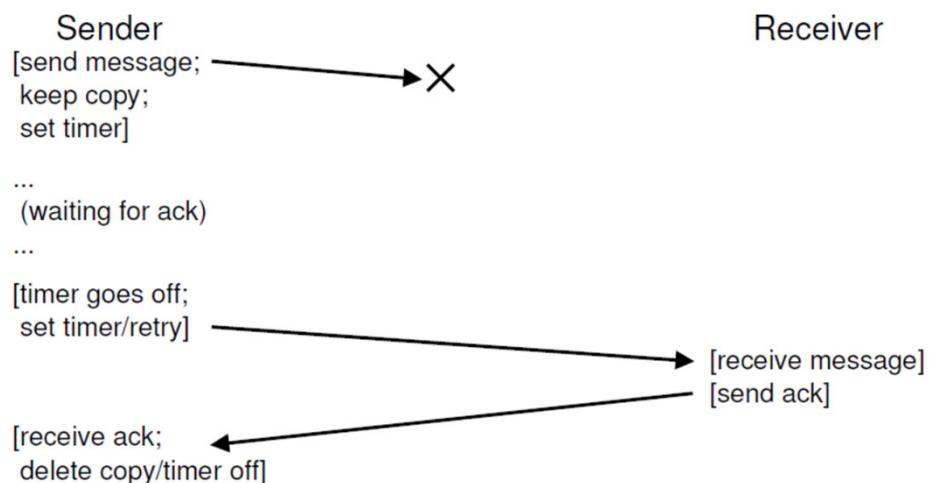
- נרצה להבין את העקרונות הבסיסיים מאחוריו פרוטוקול IP/TCP.
- הבעיה הראשונה שנרצה להתגבר עליה היא אובדן נתונים.
- הרעיון הבסיסי הוא פשוט: הצד מקבל ישלח אישור acknowledgement או בקיצור ack (צד השולח עם קבלת ההודעה ממנו).
- אם הצד השולח קיבל ack, הוא יכול להיות סגור וב吐וח שההודעה שלו הגיעו ליעדה.



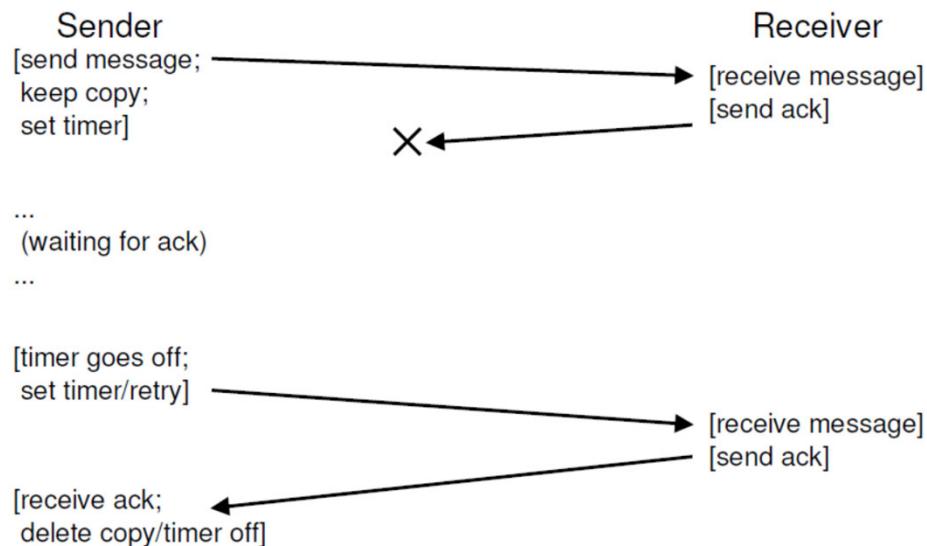
ומה אם ההודעה הלכה לאיבוד?

- השולח לא ממתין לנ匝ח כי אול' ההודעה לא הגיעו למקבל.
- במקומות זאת, השולח קובע timeout – פרק זמן שבו הוא ממתין לקבלת ack מהתיכון המתקבל.
- במידה ובפרק הזמן הנ"ל השולח לא קיבל ack, הוא קובע כי ההודעה הלכה לאיבוד.
- במקרה זה, השולח פשוט מנסה לשלוח שוב את ההודעה.
- שימוש לבב: השולח חייב לשמור אצלו את ההודעה המקורי עד לקבלת ack.

timeout/retry



ומה אם ה-ack הלך לאיבוד?



sequence number

- כאשר ack הולך לאיבוד, אז:
 - מבחינת השולח, המצב נראה כמו במצב שבו ההודעה עצמה הלכה לאיבוד; لكن השולח ישלח שוב את ההודעה.
 - מבחינת המקלט, יש בעיה – הוא עלול לקבל את אותה הודעה **פעמיים!**
 - לא היitem רוצים שפוקודת העברה בנקאית לחשבו אחר תישלח פעמיים, נכון?
- כדי שהמקבל יוכל לזהות הודעות כפולות, השולח מעניק לכל הודעה מספר ייחודי, למשל, מספר סידורי שמתחל ערך כלשהו וגדל בכל הודעה חדשה.
- כך המקלט יכול "לזרוק" הודעות שהוא קיבל פעמיים.

TCP/IP מרכיב הרבה יותר

- exponential backoff – איך לקבע את ה-timeout בצורה דינמית?
- congestion control – איך למנוע עומסים ברשת?
- flow control – המקלט מודיע לשולח כמה מידע הוא יכול לקבל.
- קודים לגילוי ותיקון שגיאות.
- אפשרות להצפנה המידע.
- ועוד מאות אופטימיזציות ופתרונות...

תכנות מונחה אירועים

החומר בפרק זה מבוסס על פרק 33 (Event-based Concurrency) מהספר OSTEP.

הבעיה: שרת מקבילי

- שרתים אמיטיים צריכים לטפל בהרבה לוחות בו-זמןית.
 - לעיתים אפילו עשרות או מאות אלפיים.
- גישה 1#: עבור כל לוח, השרת יקצה חוט שיטפל בבקשות שלו.
- 诒רונות:
 - תכונות פשוט ונוח. (רק אם הוגדר).
 - ניצולiesel של מערכות רבות מעבדים.
- חסרונות:
 - כל חוט צריך לא מעט זיכרון (כמה MB בערך).
 - תקורה של החלפות הקשר בין החוטים.
- גישה 2#: תכונות מונחה אירועים.

לולאת אירועים

server pseudocode:

```
while (1) {  
    events = getEvents();  
    for (e in events)  
        processEvent(e);  
}
```

השרת ממתין לאירועים
כמו התגובות של לקוחות
חדים או בקשות חדשות
מצד לקוחות קיימים.

כאשר מתבצעים אירועים, השרת מטפל בהם בזיה
אחר זה.

קריאה המערכת (select())

```
int select(int nfds,  
          fd_set *readfds, fd_set *writefds,  
          fd_set *errorfds,  
          struct timeval *restrict timeout);
```

ARGINENTIM:

- **readfds** – קבוצת FDs לקרואיה.
- **writefds** – קבוצת FDs לכתיבה.
- **errorfds** – אנחנו נתעלם מהארגומנט זהה ע"י העברת **NULL**.
- **timeout** – זמן המתנה (**NULL** עבור המתנה אינסופית).
- **nfds** – מספר ה-FD המקורי שיידק בכל אחת מהקבוצות הנ"ל.

קריאה המערכת (`select()`)

- פעולה: המתנה עד שאחד ה-FDs באחת הקבוצות או `readfds` או `writefds` מוכן לקריאה או לכתיבה (בהתקמה), או עד אשר חלף פרק הזמן המצוין ב-`timeout`.
 - אם קיימוד FD מוכן לקריאה אז פועלות(`read` עליה) תחזור מיד ולא תחסום את התהיליך.
 - אם קיימוד FD מוכן לכתיבה אז פועלות(`write` עליה) תחזור מיד ולא תחסום את התהיליך.
- ערך חוזר: מספר ה-FDs המוכנים בכל הקבוצות.
 - בנוסף, הקבוצות שהועברו כארגוניים יעדכנו כך שיצביעו על ה-FDs המוכנים.

דוגמת קוד עם select()

```
int main(void) {
    // the server listens to a port
    // and then accepts a bunch of sockets (not shown)

    while (1) {
        // initialize the fd_set to all zero
        fd_set readFDs;
        FD_ZERO(&readFDs);

        // set the bits for the descriptors this server
        // is interested in (all from min to max)
        for (int fd = minFD; fd < maxFD; fd++)
            FD_SET(fd, &readFDs);
```

דוגמת קוד עם select()

```
// do the select
int rc = select(maxFD, &readFDs,
                  NULL, NULL, NULL);

// check which have data using FD_ISSET()
for (int fd = minFD; fd < maxFD; fd++)
    if (FD_ISSET(fd, &readFDs))
        handle_request(fd);
}
```

פונקציית הטיפול בלקוח,
נניח כי היא ממומשת בקבץ אחר

וסרנות השימוש ב-`select()`

- דוגמת הקוד שהבנו משתמש בחוט אחד בלבד ולכן לא יכולה לנצל מעבדים רבים ליבوت.
- מימושים חכמים יותר יפזרו את הבקשות בין חוטים שונים של אותו מאגר (thread pool).
- אם הטיפול בבקשת – כלומר הפונקציה `()` `handle_request` בדוגמה הקוד – קורא לפועלה חסימת (לדוגמה פתיחת קובץ מהדיסק) אז השרת יכול עלול להיחסם.
 - שוב בעית נצילות בגל שלא השתמשנו בחוטים.
- המימוש של `set_fd` מוגבל לקבוצות בגודל מקסימלי של 1024.

תרגול 10

למה צריך זיכרון וירטואלי?

Paging במעבדי אינטל 32-ビト

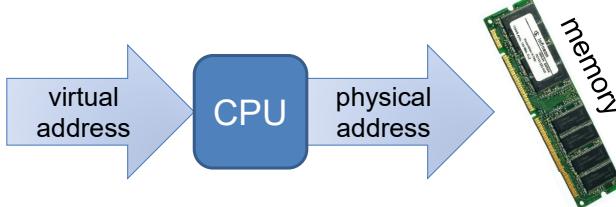
Paging במעבדי אינטל 64-ビト

TL;DR

- גישה ישירה לדיסקון הפיזי הייתה יוצרת הרבה בעיות: מחסום בזיכרון רציף, היעדר בידוד בין תהליכיים, מגבלת על מרחב הזיכרון האפשרי.
- האבסטרקציה שפותרת את כל הבעיה הללו היא **זיכרון וירטואלי**.

process A:

```
...
mov $0x700, %rax
add %rax, %rbx
...
...
```



- אבל אין ארוחות חינם: **זיכרון וירטואלי פוגע בовичעים**.
- כל פקודה גישה לדיסקון דורשת תרגום יקר: וירטואלי → פיזי.

*"All problems in computer science can be solved by another level of indirection.
... except, of course, for the problem of too many indirections."*

-- David Wheeler

למה צריך זיכרון ורטואלי?

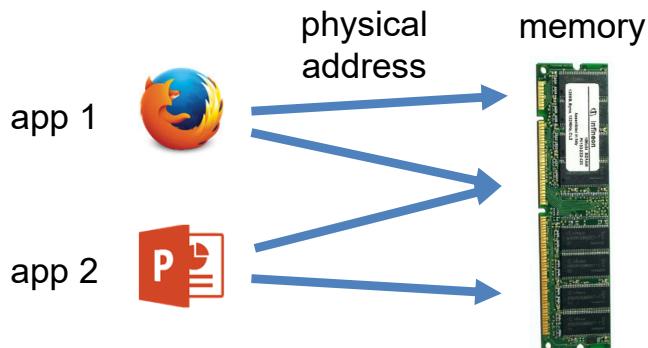
או: מדוע לא ניתן ישירות לזיכרון הפיזי?

זיכרון פיזי

- התקני האיחסון במחשב נחלקים לשניים:
 - זיכרון (DRAM) – איחסון נדייף, קטן יותר ומהיר יותר (סדר גודל 100ns).
 - דיסק קשיח – איחסון עמיד, גדול יותר ואיטי יותר (סדר גודל 1ms).
- פקודות מכונה יכולות לפעול רק על רגיסטרים ו/או נתונים **בזיכרון**.
 - הקוד המבוצע ע"י המעבד והנתונים הדרושים לביצוע הקוד חייב להיות בזיכרון בזמן הביצוע.
 - אם רוצים לעבוד מידע מהdisk, יש להביא אותו לזכרון, לעבוד אותו, ולכטבו את התוצאה חזרה לדיסק.
- הגישה לזכרון היא, בסופו של דבר, באמצעות **כתובות פיזיות** של בתים (bytes). למשל, עבור זיכרון בגודל 8GB הכתובות הפיזיות הן מספרים שלמים בתחום $[1 - 8G]$.
- עם זאת, יש חסרונות רבים לגישה באמצעות כתובות פיזיות.

נדיף (volatile) – נמוך ללא אספקת מתח, למשל עם כיבוי המחשב.

cosa #1: היעדר בידוד/הגנה בין תהליכיים



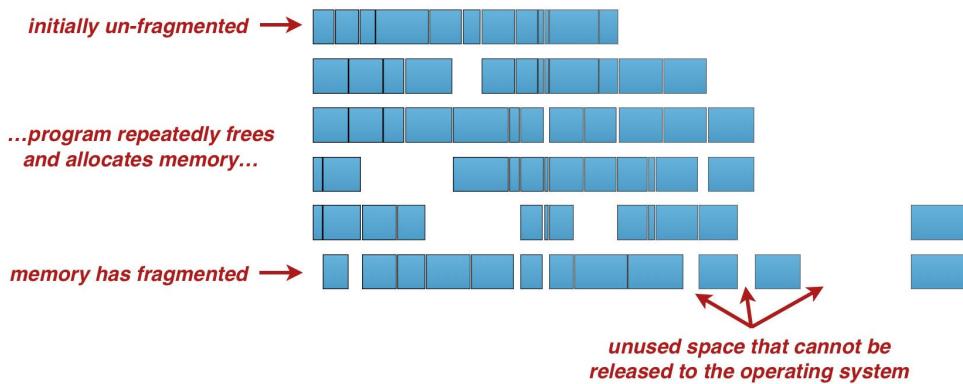
- אין הגנה על המידע – תהליך א' יכול לגשת לזיכרון של תהליך ב'.

Without virtual memory, application 1 references the memory with a physical address, and application 2 does the same.

So they might access the same memory region intentionally or unintentionally.
We would like to protect the processes from each other and isolate them.

הסרון #2: מחסור בזיכרון רציף

- במערכת אמיתית, הזיכרון עובר קיטוע (fragmentation):



- גם כאשר יש מספיק זיכרון במערכת, הוא "שבור" לרשאים.

מתוך:

<https://collectiveidea.com/blog/archives/2015/02/19/optimizing-rails-for-memory-usage-part-2-tuning-the-gc>

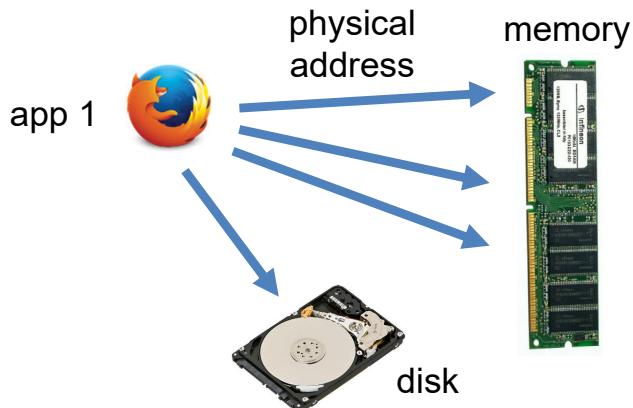
פרגמנטציה (fragmentation)

- בזיכרון זיכרון כתוצאה מאופן שימוש לא יעיל.
- יש שני סוגי פרגמנטציה:
 - **פרגמנטציה חיצונית** (external fragmentation) – בזיכרון מוחוץ למקטעי הזיכרון בגאל שהם מפוזרים למרחב.
 - לדוגמה: מערך C חייב להיות מוקצה בצורה רציפה בזכרון.
 - ניתן כי לא ניתן להקצת מערך בגודל נתון למורות שיש מסויק זיכרון – סכום כל החורים למרחב גדול מסויק, אבל החורים לא מאורגנים באופן רציף.
 - **פרגמנטציה פנימית** (internal fragmentation) – בזיכרון בתוך מקטעי הזיכרון כתוצאה מהקצתה יתר.
 - לדוגמה: מהדר מסויים מיישר כל הקצתה זיכרון לכפולה של בלוק בגודל N.
 - אם המשתמש מבקש זיכרון בגודל $> N$, שאר הזיכרון יתbezבז.

פתרון אפשרי לפרגמנטציה חיצונית: דחיסה (compaction) שתזיז את מקטעי הזיכרון למרחב כדי לאחות אותם לבлок אחד רציף.

פתרון אפשרי לפרגמנטציה פנימית: הקצתה זיכרון גמישה יותר. אבחנה חשובה שנראה בהמשך: פרגמנטציה יכולה להתפרש הן למרחב הזיכרון הווירטואלי והן למרחב הזיכרון הפיזי.

הסרון #3: מגבלת זיכרון

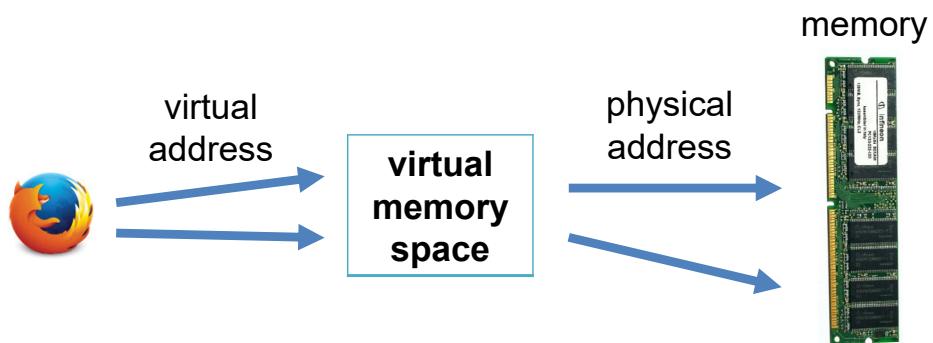


- היאנו רוצים להשתמש גם בשטח האחסון שקיים בדיסק,
בצורה שקופה לקוד האפליקציה.

Another problem with referencing the memory directly is that the application might need more memory than available, so we would like to use the disk to back up some of the application data.

הפתרון: זיכרון וירטואלי

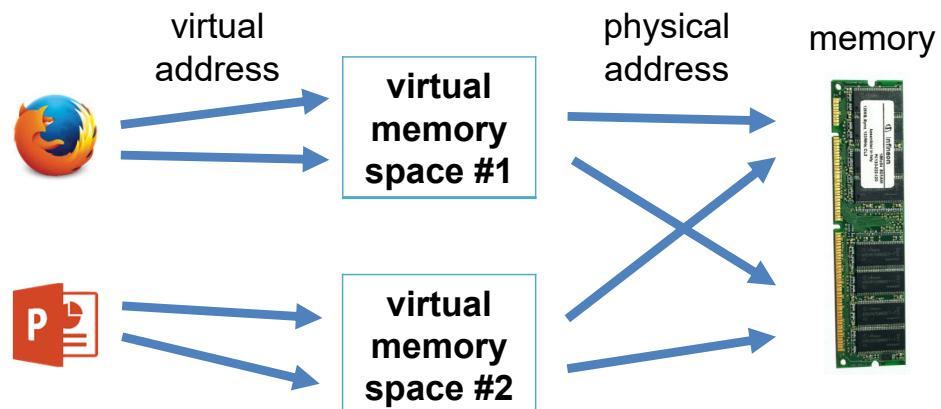
- פקודות המכונה ייגשו אך ורק לכתובות זיכרון וירטואליות.
- מערכת הפעלה תגדיר מייפוי (= פונקציה) בין כתובות וירטואליות לפיזיות: לכל כתובה וירטואלית מתאימה בדיק כתובה פיזית אחת.
- המעבד יתרגם כתובה וירטואלית \rightarrow פיזית בזמן הגישה.



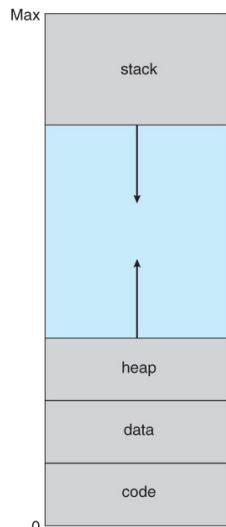
. MMU = memory management unit נקרא על התרגום האחראי במעבד.

זיכרון וירטואלי נתן בידוד/הגנה

- תהיליך יכול לגשת רק למרחב הזיכרון הוירטואלי שלו עצמו.
- כל תהיליך מקבל אשלייה שהוא לבד במערכת.



זיכרון וירטואלי מספק רציפות



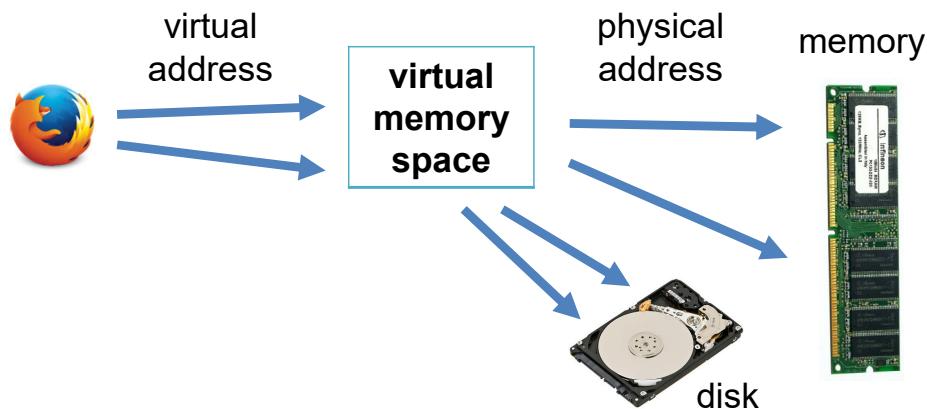
- תהליך חדש מקבל מרחב זיכרון וירטואלי "נקי" ורציף.
- בנוסף, מרחב הזיכרון הווירטואלי של תהליך יכול להיות גדול הרבה יותר מהזיכרון הפיזי הזמין.
- ← מערכת הפעלה תוכל למצוא בקלות יותר זיכרון רציף במרחב הווירטואלי.
- שימוש לב: הזיכרון הפיזי המתאים לא חייב להיות רציף!

התמונה מתוך:

https://www2.cs.uic.edu/~jbell/CourseNotes/OperatingSystems/9_VirtualMemory.html

זיכרון וירטואלי מאפשר swapping

- ניתן למפות חלקים מהזיכרון הוירטואלי אל הזיכרון או אל הדיסק.
- ← המשתמש יראה יותר זיכרון ממה שיש באמת במערכת.



And if the application requires more memory than available in the DRAM, or if we have many processes that require more memory than available, the operating system can let the disk back regions of virtual memory.

זיכרון וירטואלי מציע יתרונות נוספים

- **demand paging** – חסכו של זיכרון פיזי ע"י הקצתתו רק בגישה הראשונה לזכרון הווירטואלי.
 - למשל: אם הקצנו מערך גדול באמצעות (`malloc` ולא ניגשנו לחלקים ממנו, החלקים האלה לא יהיו מגובים בזכרון הפיזי).
- **deduplication** – חסכו של זיכרון פיזי במידה ואפליקציות שונות משתמשות באותו מידע **לקראיה בלבד**.
 - למשל, מרבית התהליכים משתמשים במידע של ספריית `cbs` (לקראיה בלבד).
 - לכל תהליך מרחב זיכרון וירטואלי שונה, אבל ככל יכולם למפותו לאזור פיזי שבו ישבת הספרייה `cbs`.
- **copy-on-write** – מנגנון לחיסכון של זיכרון פיזי ולמנוע העתקות מידעת מיותרות – נראה בתרגול הבא.
- ועוד יתרונות רבים אחרים...

PAGING במעבדי אינטל 32-ビト

או: איך מממשים זיכרון וירטואלי?

דפים ומסגרות

מרחב הזיכרון הווירטואלי

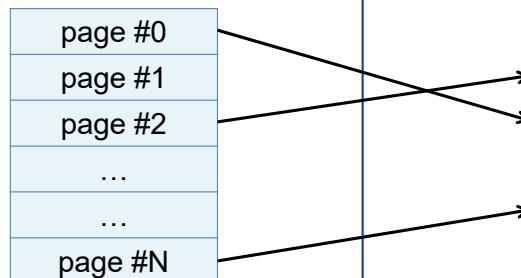
- מחולק לדפים (**pages**).
- גודל דף == גודל מסגרת (4KB).
- הדפים מושרים בזיכרון הווירטואלי.

page #0
page #1
page #2
...
...
page #N

מרחב הזיכרון הפיזי

- (frames) בלוקים עיקריים בגודל קבוע (4KB בארכיטקטורת IA-32).
- המסגרות מושرات בזיכרון הפיזי.

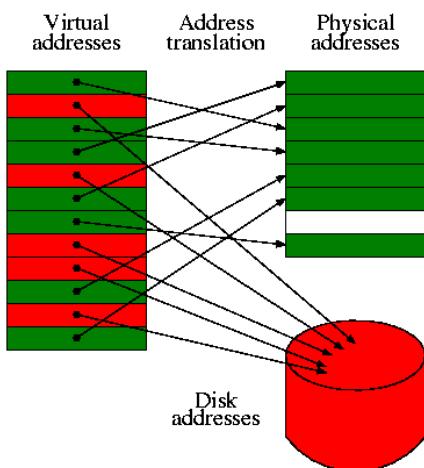
frame #0
frame #1
frame #2
...
...
frame #M



יש הרבה שיטות לממשק זיכרון וירטואלי.

אנו נלמד על שיטת **paging**, כי זו השיטה לימוש זיכרון וירטואלי במעבד אינטל. כאן כדאי לזכור קצר על הרזולוציה של תרגום וירטואלי→פיזי ולהסביר למה תרגום ברמת הבית הבודד אינו אפשרי.

לא כל הדפים ממופים למסגרות פיזיות!



- חלק מהדפים לא ממופים כלל, כלומר הדף לא מגובה בזיכרון ולא בדיסק.
- מטעמי חיסכון בזיכרון ובזמן, אין טעם להקצת מרASH את כל המרחב הווירטואלי של תחילה.
- חלק מהדפים יכולים להיות מגובים בדיסק.
- נאמר כי הדפים **swapped out**.
- פרטיהם נוספים בתרגול על מטען הדפים.

דפים ומסגרות בארכיטקטורת 32-AI

- בארQUITקטורת 32-AI (ארQUITקטורת 32 בית של אינטלי):
 - ה זיכרון הווירטואלי הוא ברוחב 32 בית.
 - ה זיכרון הפיזי הוא ברוחב 32 בית.

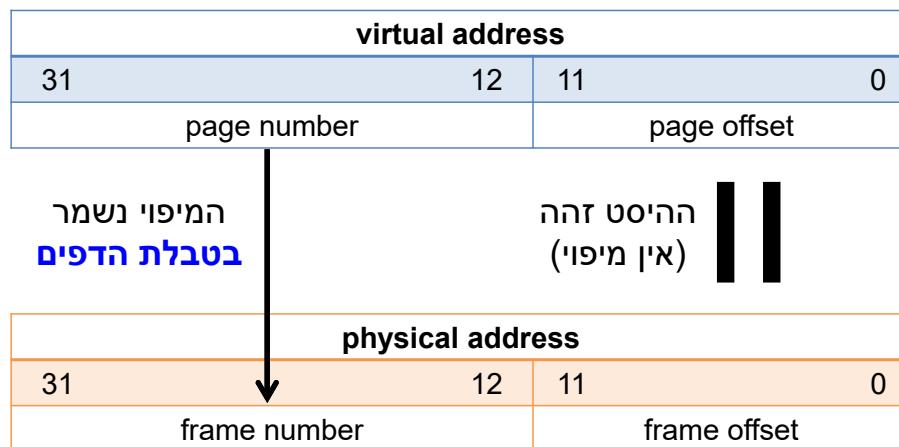
- מה מספר הדפים במרחב הווירטואלי?

$$\frac{\text{space size}}{\text{page size}} = \frac{2^{32}}{2^{12}} = \frac{4\text{GB}}{4\text{KB}} = 1\text{M} \cong 1,000,000$$

- מה מספר המסגרות במרחב הפיזי?
- כג"ל (הчисלוב זהה).

מייפוי דפים למסגרות

- בכל גישה לזיכרון, המעבד מתרגם את הכתובת הווירטואלית לכתובת פיזית באופן הבא:





טבלת הדפים (page table)

- לכל תהילך יש טבלת דפים משלו – **מבנה נתונים אשר ממפה בין דפים למסגרות**.
- ניתן למשם טבלת דפים באמצעות מבנים נתונים שונים: מערך פשוט, עצים, טבלאות גיבוב (hash tables), ...
- עבור כל דף במרחב הזיכרון הווירטואלי של התהילך, יש כניסה בטבלת הדפים אשר מצינית:
 - האם הדף נמצא בזכרון ובאיזה מסגרת?
 - האם הדף נמצא בדיסק ובאיזה מיקום?
 - האם הדף מעולם לא הוקצה? (כלומר אינו בזכרון ואיןנו בדיסק)
- טבלת הדפים אחראית לתפקידים נוספים כמו הגנת גישה.
- **למשל:** טבלת הדפים מסמנת דפים לקריאה בלבד ומונעת גישות כתיבה.

ניסון 1#: טבלת דפים ליניארית

- מערך שבו הכניסה $\text{-}k$ מכילה את המיפוי של הדף $\text{-}k$.

#0	
#1	
#2	
...	
...	
#k	
...	
...	
...	
#N	



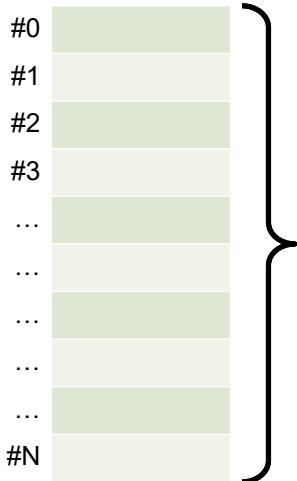
כמה ביטים?

כמה ביטים?

תשובות בשקף הבא.

ניסון 1#: טבלת דפים ליניארית

- מה גודל טבלת הדפים?



- מספר הכניסות במערך הוא כמספר הדפים:
 $N=1 M$
 - נניח שכל כניסה במערך היא בגודל 4 בתים:
20 עבורי מספר המסגרת
+ 12 עבורי סיביות בקרה
- ↳ גודל טבלת הדפים הוא 4MB
(ולכל תהליך טבלת דפים משלה!)

בעיקרון ניתן היה להסתפק בפחות מ-12 ביטים עבור סיביות הבקרה.
אבל משיקולים הנדרשים, נוח יותר לעבוד עם גודל כניסה שהוא חזקה של 2, ולכן מעגלים למעלה כך שה כניסה תהיה בגודל 32 ביט.

ניסון 1#: טבלת דפים ליניארית

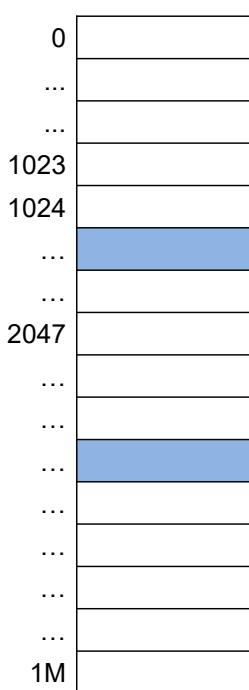
- עברו 100 תהליכי, התקורה על הזיכרון הפיזי היא 400 MB.
- בפועל, תהליכי קטנים (ויש הרבה כאלה) ניגשים רק לחלק קטן של מרחב הזיכרון הוירטואלי, ולכן זה בזבזני להחזיק את הטבלה כולה.
- יתרה מזאת, טבלת דפים ליניארית פשוט אינה ישימה בארכיטקטורות חדשות.
- למשל, נניח כי רוחב של כתובות וירטואלית וфизית הוא 48 ביטים.
- מרחב הזיכרון הוירטואלי הוא בגודל $TB = 2^{48}$.
- גודל דף הוא 4KB ולכן יש $2^{36} = 2^{12}/2^{48}$ כניסה במערך.
- כל כניסה היא בגודל 8 בתים.
- ← גודל טבלת הדפים יהיה **512GB** לכל תהליך!

הפטקה

ACCESSING YOUR MEMORY DURING THE EXAMS



ניסון 2#: טבלת דפים היררכית



- נניח כי ההליך מסויים P ניגש לשני דפים בלבד.
- טבלת הדפים הליניארית של ההליך P משתמשת בשתי כניסה בלבד, כפי שקרה בתרשימים:
- קל לראות את בזבוז הזיכרון...

ניסון 2#: טבלת דפים היררכית

- נחלק את הכניסות בטבלה לבלוקים בגודל מסגרת פיזית.

- כמה כניסה יהיו בכל בלוק?

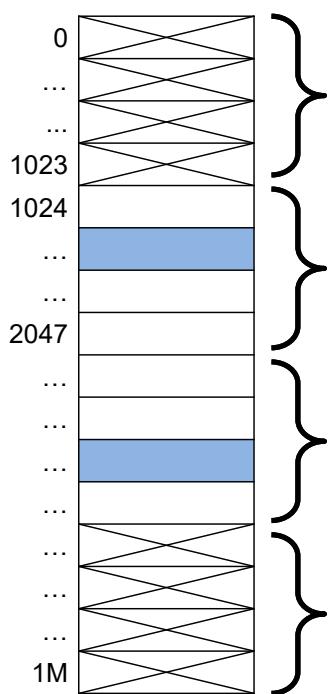
$$\frac{\text{frame size}}{\text{entry size}} = \frac{4\text{KB}}{4\text{B}} = 1024$$

- כמה בלוקים יהיו?

$$\frac{1\text{M}}{1024} = 1024$$



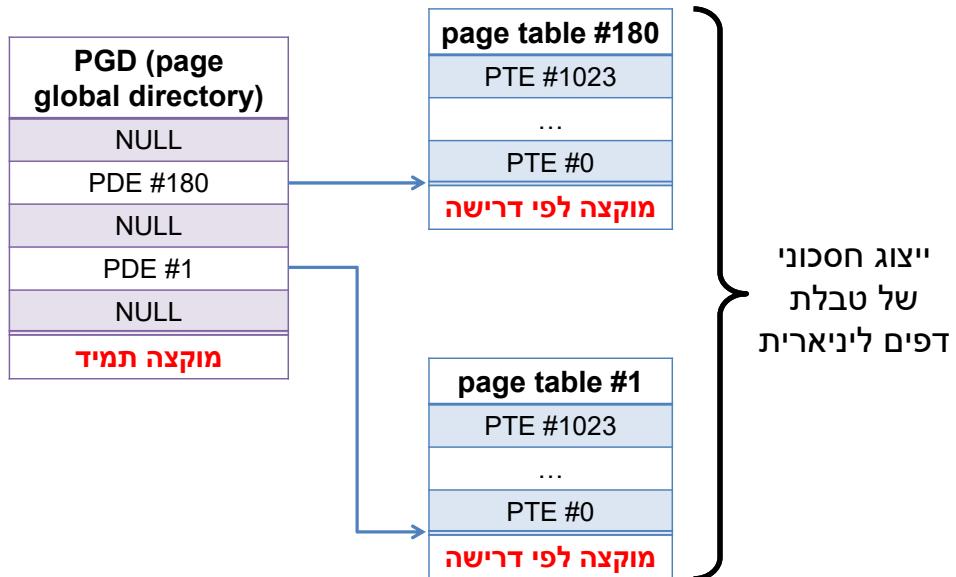
ניסון 2#: טבלת דפים היררכית



- נשים לב כי מרבית הבלוקים ריקים לאמרי...
- ולכן לא כדאי להקצות אותם.

- נשמר טבלה נוספת עם 1024 כנישות אשר תצביע לבלוקים:
 - NULL עבור בלוק ריק.
 - כתובת פיזית עבור בלוק מוקצה.

ניסוח 2#: טבלת דפים היררכית



PDE = page directory entry

PTE = page table entry

תהליך תרגום כתובות וירטואלית

- איך המעבד מתרגם כתובות וירטואלית V לכתובות פיזיות?
- בשלב הראשון, המעבד מחשב את מספר הדף הווירטואלי:

$P = V(12) >>$

- בטבלת דפים ליניארית:

1. המתרגoms נמצא בכינוי P במערך.

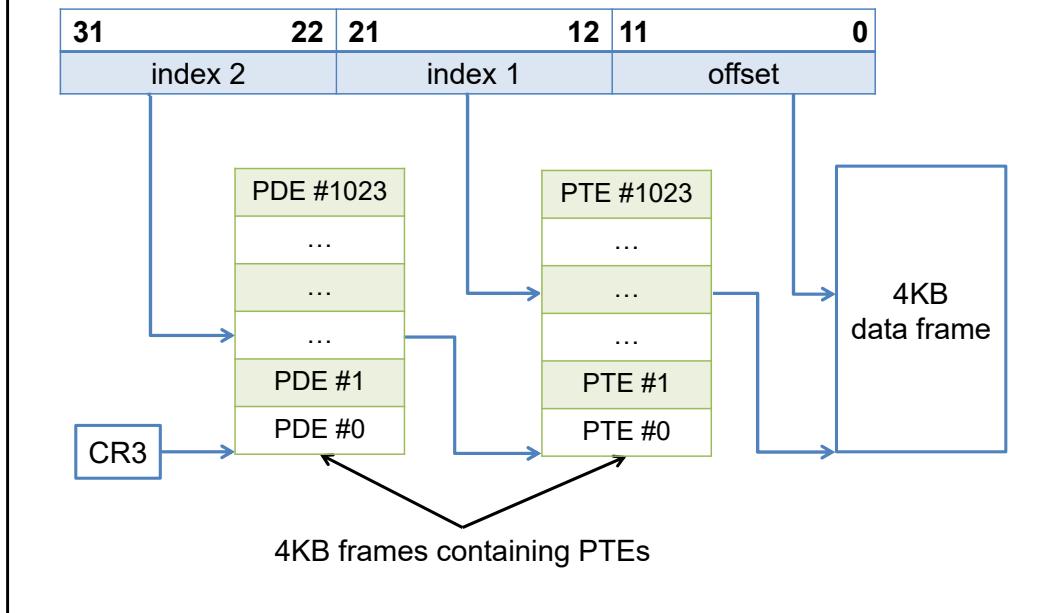
- בטבלת דפים היררכית:

1. המעבד קורא את הכינוי $1024 / P$ בرمאה העליונה של העץ.

2. אם הכינוי הוא $= NULL$, אז אין תרגום (הדף לא בזיכרון).

3. אחרת, המתרגoms נמצא בכינוי $1024 \% P$ בرمאה התחתונה של העץ.

פירוק כתובת וירטואלית לשדות

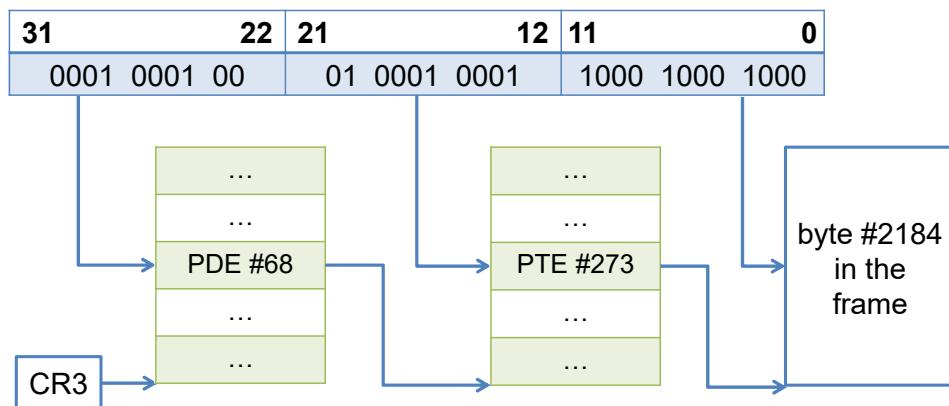


The divide-by-1024 and modulo-by-1024 operations are equivalent to simple shift operations because 1024 is a power of two.

(Think of divide-by-10 and modulo-by-10 in a decimal base.)

דוגמה: פירוק כתובת וירטואלית לשדות

- ניקח לדוגמה את הכתובת 0x11111888 .
- נכתבת הכתובת בסיסי בינהרין ופרק אותה לשדות:



טבלת הדפים בארכיטקטורת 32-A

- בגלל הביעות בטבלת הדפים הליניארית, אינטלי בחרה בטבלת דפים היררכית בצורה עץ (דיליל) עם שתי רמות.
- מבנה הנתונים: `tree radix` במקום מערך.
- הרמה התחתונה בעץ שומרת מיפויים בין דפים למסגרות---בדיקה כמו במערך.
- הרמה העליונה בטבלת הדפים מצביעה למסגרות של הרמה התחתונה.
- כל 1024 כניסה סמוכות ברמה התחתונה ישמרו במסגרת נפרדת (המסגרות לא בהכרח רציפות בזיכרון הפיזי, בניגוד למערך).
- במידה ואף אחת מהכニיות ברמה התחתונה לא מפנה דף, אין צורך להקצות מסגרת ברמה התחתונה.
- כאשר מוקצה דף חדש לשימוש התהיליך, צריך להקצות, לפי הצורך, מסגרות עבור הרמות בהיררכיה עד (לא כולל) השורש.
- רגיסטר מיוחד בשם **CR3** מצביע לשורש טבלת הדפים של התהיליך הנוכחי.

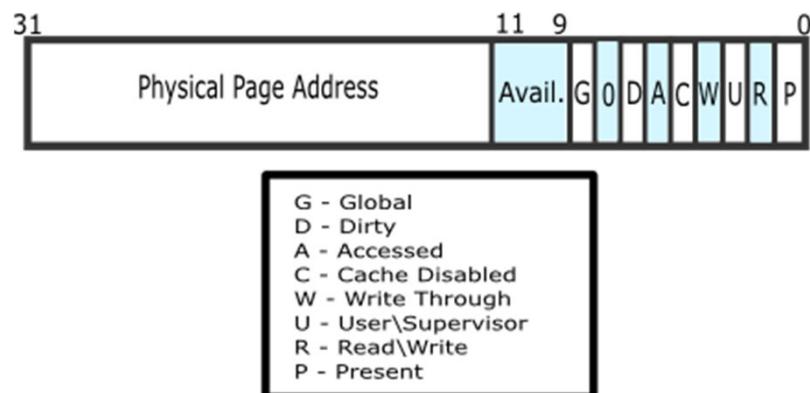
האם CR3 מכיל כתובת וירטואלית או פיזית?

רגיסטר CR3 חייב להצביע לכתובת המסגרת הפיזית. אחרת, אם רגיסטר CR3 היה מכיל כתובת וירטואלית, תהיליך תרגום הכתובות (page walk) היה נקלע לרוקורסיה אינסופית.

מבנה כניסה בטבלת הדפים

- כניסה ברמה הראשונה נקראת **PDE** = page directory entry.
- כניסה ברמה השנייה נקראת **PTE** = page table entry.
 - בפועל, קוראים לכל הכניסות בכל הרמות PTE.
 - כל כניסה בטבלת הדפים היא בגודל bit 32.
- המידע שכינסה מכילה תלוי בביט 0 של ה-PTE), המציין האם הדף נמצא בזיכרון הראשי.
 - 1 == present: הדף נמצא בזיכרון הפיזי.
 - 0 == not present: הדף לא נמצא בזיכרון הפיזי.

Page Table Entry



כניסה בטבלת הדפים, כאשר $\text{present} == 1$

- **מספר המסגרת** בה מאוחסן הדף.
- 20 ביטים, כאשר כתובות זיכרון פיזי באורך 32 ביט.
- **בית accessed** (נקרא גם בית referenced): מודלק ע"י החומרה בכל פעם שמתבצעת גישה לכתובת בדף. בית זה מכובה באופן מחזורי ומשמש למידניות פנוי הדפים לדיסק.
- **בית dirty** (נקרא גם בית modified): מודלק ע"י החומרה בכל פעם שמתבצעת כתיבה לנטוון בדף. במידה והדף שייר לקובץ (לדוגמה) יידע שיש לכתוב אותו **חרזה לדיסק** מתיישר.
- **בית read/write**: הרשות גישה.
 - 0 = קרייה בלבד. 1 = קרייה וכתיבה.
- **בית user/supervisor**: גישה מיוחסת.
 - 0 = גישה לקוד הגרעין בלבד. 1 = גישה לכל תהיליך.

הרשאות הדף נקבעות לפי הרשותות האזר (נראה בתרגול הבא) בהתאם לכללים הבאים:

- אם יש הרשות write באזר, מודליקים את w/z (הכל מותר).
- אחרת, אם באזר יש הרשות read או execute, מכבים את w/z (モוטר רק לקרוא).

כניסה בטבלת הדפים, כאשר $\text{present} == 0$

- יש שתי אפשרויות:
 - .1 אם כל הביטים ב-PTE הם אפס, אז הדף לא ממופה כלל למרחב הזיכרון של התהליין.
 - .2 אחרת, אם לפחות אחד מ-31 הביטיםعلויונים שונה מאפס, אז הדף נמצא במאגר דף-דף (swap area) בדיסק, וב-PTE נשמר את כתובתו ב-swap area.

TLB - Translation Lookaside Buffer

- תרגום כתובת וירטואלית **לפיזית** קורה כל גישה לזיכרון.
- 50%--30% מהפקודות בתכנית מ모יצעת ניגשות לזיכרון ↲ תקורה גבוהה.
- כדי לשפר את הביצועים, מעבדי אינטל מכילים מטמון (cache) מיוחד, ה-**TLB**, אשר מכיל את התרגומים האחרונים בהם השתמשו.
- המעבד מחפש ב-TLB לפני החיפוש בטבלת הדפים. אם התרגום המבוקש נמצא ב-TLB, נחסכו גישות יקרות לזיכרון (מספר הרמות בהיררכיה).
- אם התרגום המבוקש לא נמצא ב-TLB, המעבד פונה לחיפוש בטבלת הדפים ואז מוסיף ל-TLB את התרגום החדש (לטובת הגישות הבאות לזכרון).

page number	frame number	flags
12	25	r/w, accessed
55	93	accessed, dirty
...	...	

שימוש לב: ה-TLB, בניגוד לטבלת הדפים, אינו יושב בזיכרון הפיזי אלא בזיכרון ייעודי ומהיר בתוכן המעבד.
לכן חיפוש ב-TLB מהיר יותר מאשר חיפוש בטבלת הדפים.

“On average, roughly half of the instructions reference memory.”

From: <http://www.jaleels.org/ajaleel/publications/SPECanalysis.pdf>

פונקציית TLB

- ה-TLB מכיל עותק חלקו של המידע הנוכחי בטבלת הדפים, ולכן מערכת הפעלה אחראית לשמר על קויורנטיות המידע ב-TLB.
- הגרעין חייב לפוסול (invalidate) את תוכן ה-TLB במקרה מסוים,

- כasher הגרעין מוחק כניסה בטבלת הדפים (כדי לפנות מסגרת מהזיכרון לדיסק), הוא מוחק גם את הכניסה המתאימה ב-TLB.
 - אחרת, התהילה עלול לגשת למידע לא מעודכן, בעוד שה-TLB עדין מצביע למסגרת שכבר פונתה מהזיכרון.
- בעת החלפת הקשר, הגרעין מוחק את תוכן ה-TLB כולו.
 - אחרת התהילה הבא לביצוע ייגש למסגרות של התהילה שרצ לפנוי.

העשרה: הגרעין פוסל את כל תוכן ה-TLB באופן אוטומטי ע"י טעינת ערך חדש ל-**CR3** (מעבדי אינטל פועלם כך שכטיבה לרגיסטר CR3 גורמת לפונקציה אוטומטית של כל תוכן ה-TLB).

הימנעות מפסילת תוכן TLB

- **שאלה:** למה כדאי להימנע מפסילת תוכן TLB?

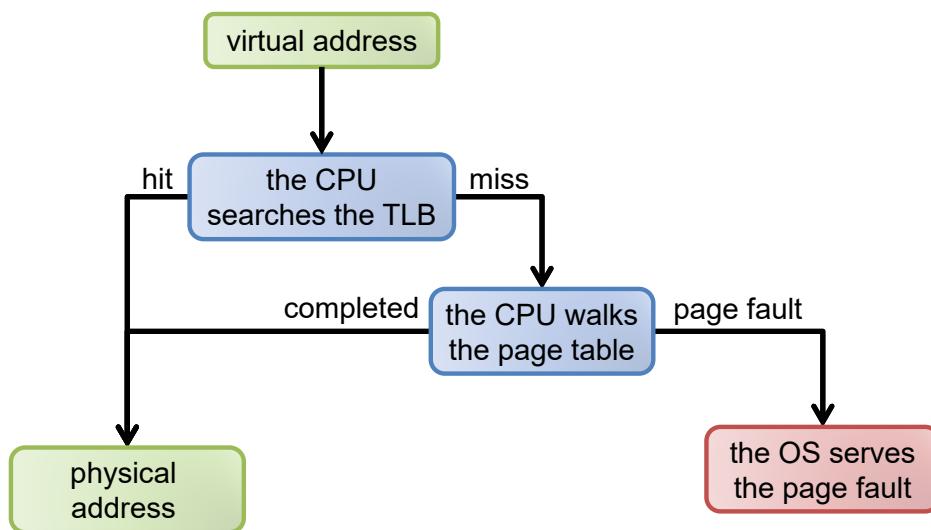
- לינוקס נמנעת מפסילת תוכן TLB בהחלפת הקשר אם:
 1. התהיליך הבא לביצוע חולק את אותו מרחב זיכרון (אותן טבלאות דפים) ייחד עם התהיליך הקודם (שני חוטים של אותו תהיליך).
 2. התהיליך הבא לביצוע הוא תהיליך גרעין (**kernel thread**).
- לתהילכי גרעין אין מרחב זיכרון שלהם, והם פועלים על מרחב הזיכרון של הגרעין.
- תהיליך גרעין מנצל את טבלאות הדפים של תהיליך המשמש שרץ לפניו, מפני שאין לו טבלאות דפים משלו.

- **שאלה:** האם הגרעין יכול לגשת למרחב הזיכרון של התהיליך הקודם?

תשובות:

- (1) כדי למנוע פגיעה בביצועים (כדי למנוע TLB misses).
- (2) תיאורטית כן, אבל זה לא אמרו לקרות. הגרעין ניגש רק למבנה הננתונים הגלובליים של הגרעין, או לזיכרון של התהיליך במידה והתהיליך ביקש זאת (למשל בקריאה מערכת, read/write).

סיכום: תהליך התרגום במעבדי אינטל



השלבים בכחול מתבצעים ע"י החומרה, השלבים באדום ע"י מערכת הפעלה.

PAGING במעבדי אינטל 64-ビト

גודל מרחב הזיכרון

- במעבדי 64 ביט של אינטל (ארכיטקטורת 64x), משתמשים בכתובות וירטואליות של 48 ביט בלבד (מתוך 64 אפשריים). מה גודל מרחב הזיכרון הוירטואלי?
- $$2^{48} B = 256 TB$$
- מה היה גודל מרחב הזיכרון אם היו משתמשים בכל 64 הביטים לייצוג כתובות?
- $$2^{64} = 16 \text{ Exabyte}$$
- מדוע, אם כן, משתמשים רק ב-48 ביט?
- עבור האפליקציות הקיימות היום, אין צורך במרחב וירטואלי גדול כל כך.

1 Exabyte = 1024 Petabyte = 1024*1024 Terabyte

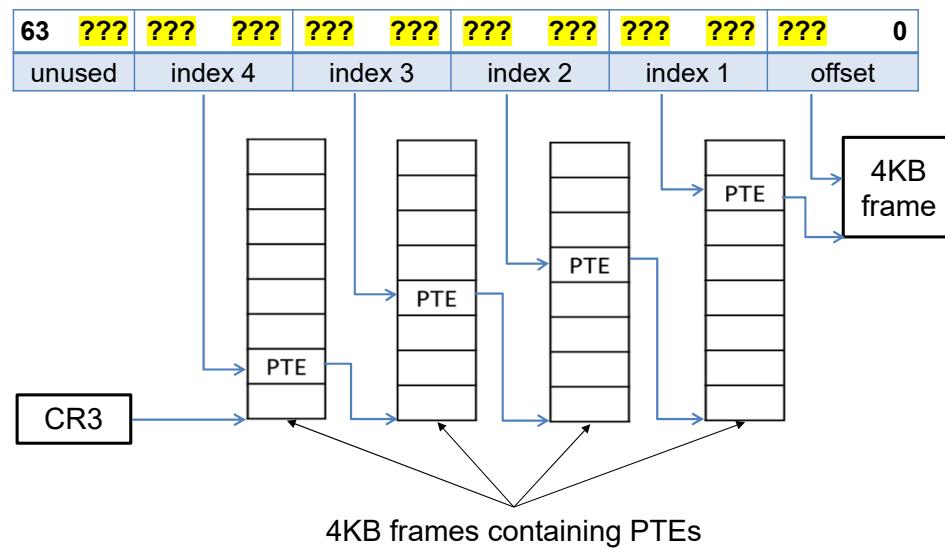
אינדקסים לטבלת הדפים

- גם בארכיטקטורת 64x גודל הדף הסטנדרטי הוא 4KB.
- גודל כניסה בטבלת הדפים הוא 8 ביטים.
- ארכיטקטורת 64x תומכת בכתובות פיזיות של עד **52** ביטים (שימו לב: מרחב הזיכרון הפיזי גדול יותר ממרחב הזיכרון הווירטואלי).
- לכן מספר המסגרת מכל $bits = 40 = 12 - 52$.
- בתוספת 12 הביטים של הדגלים והרשאות הגישה יש לנו 52 ביטים.
- גודל כניסה בטבלת הדפים הוא תמיד חזקה של 2, וכך צריכים 64 ביטים, או 8 ביטים.
- כמה כניסה (PTEs) מוכלות במסגרת (בגודל 4KB)?
$$\frac{4KB}{8B} = 512$$
- כמה ביטים צריך כדי לאנדקס אותם?
$$\log_2 512 = 9$$

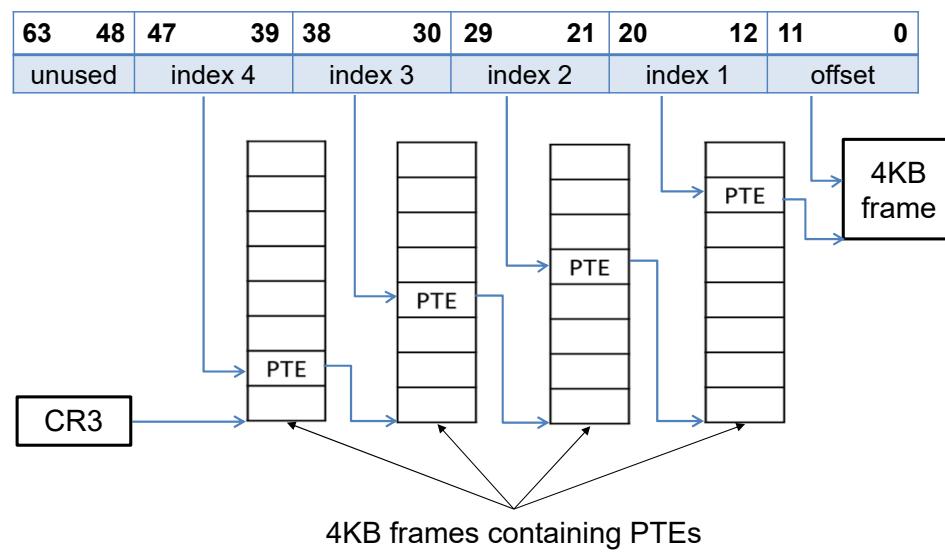
page table walk

- מעבדי 64x משתמשים בטבלאות דפים עם ארבע רמות תרגום במקומות שניים.
- לצורך הפשטות, נקרא בשם PTE עבור כניסה בכל הרמות של הטבלה.
- גודל מסגרת בכל הרמות של טבלת הדפים הוא 4KB.
- תרגום כתובות וירטואליות לפיזית נקרא page table walk כי הוא "הולך" לאורך טבלת הדפים ההיררכית.
- השלימו את הסכימה הבאה:

page table walk



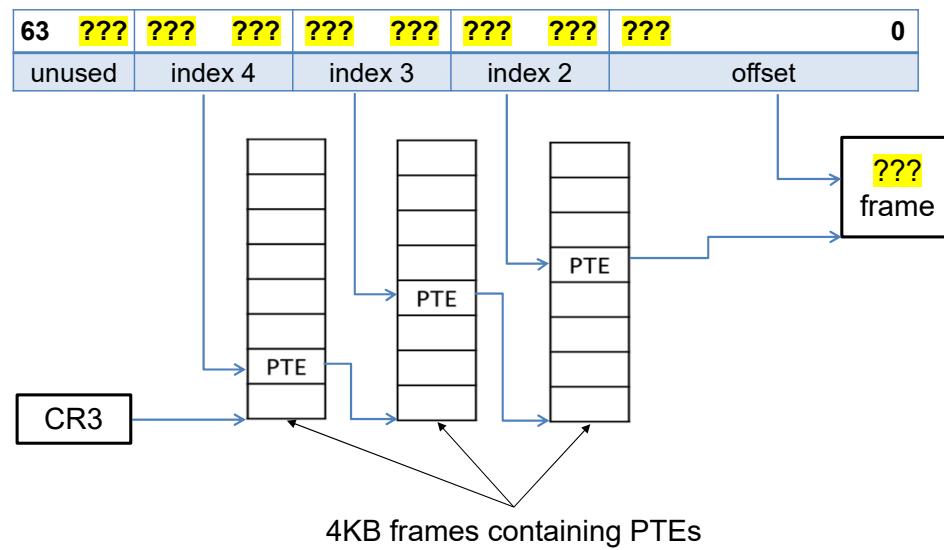
page table walk



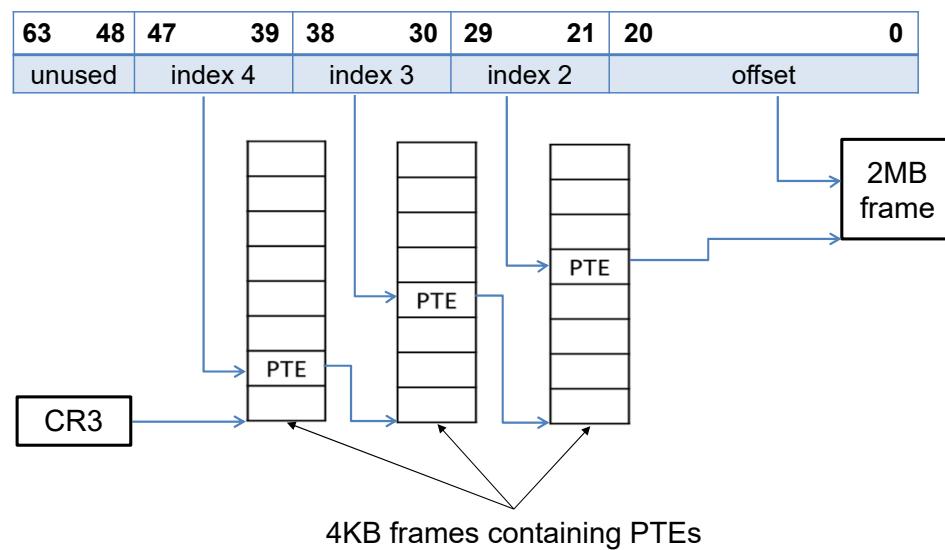
דפים גדולים

- מעבדי 64x תומכים גם בדפים גדולים (huge pages).
- תרגום של דפים גדולים "הולר" רק דרך 2 או 3 רמות של טבלת הדפים ההיררכית.
- הביטים של האינדקסים שנדרקן מצטרפים לשדה offset.
- השלימו את הסכימות הבאות ומצאו את גודל הדפים גדולים:

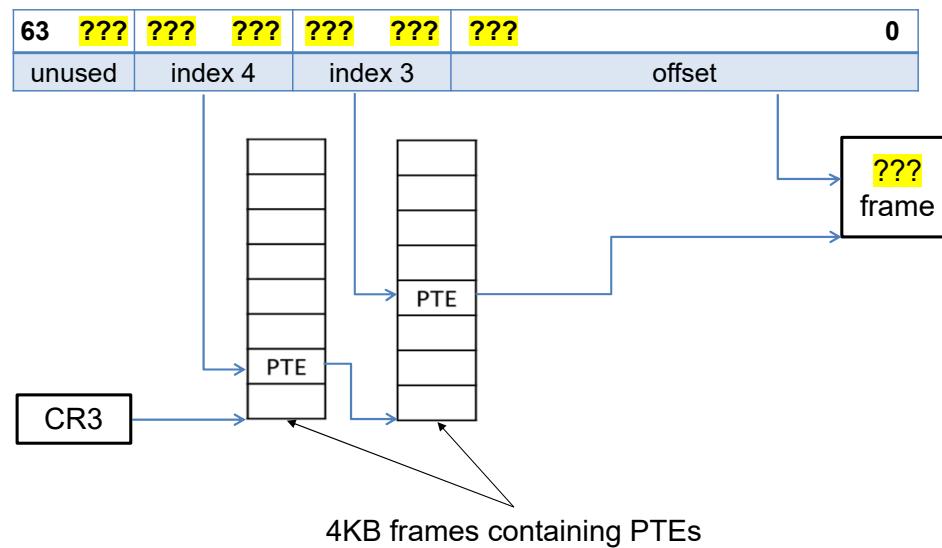
דפים גדולים



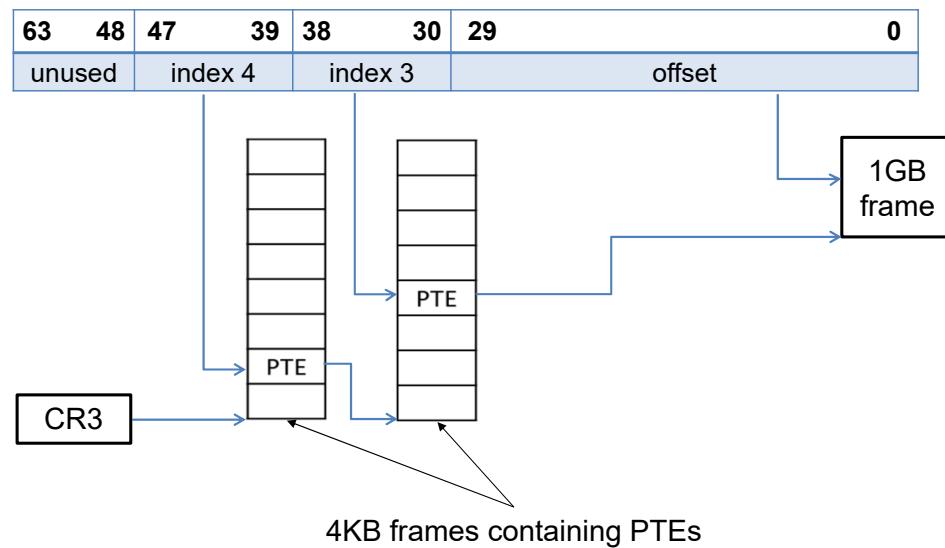
דפ'ים גדולים



דפים גדולים



דפים גדולים



יתרונות וחסרונות דפים גדולים

יתרונות

- מושפרים ביצועים (מבחינת throughput) כי הם מקטינים את התוקרה של תרגום כתובות.
- מגדילים את יעילות TLB (כל כניסה ב-TLB מכסה אזור זיכרון רחב יותר).
- מקצרים את ה-walk.page table .
- בעקביפיון, מגדילים את ייעילות caches שומרים את ה-PTEs, ועבור דפים גדולים יש פחותות (PTEs).

חסרונות

- עלולים ליצור פרוגמנטציה פנימית, ככלומר לbezch זיכרון בתוך הדפים. לדוגמה: תהליכי קטנים ידרשו 2MB+2MB במקום רק 4KB+4KB למחסנית וולרימה.
- שימוש בדפים גדולים לצד דפים קטנים יוצר פרוגמנטציה חיצונית, ככלומר מחסור בזכרון רציף.
- מערכת ההפולה תתקשה למצוא "חורים" לדפים גדולים.
- פוגעים ביצועים (מבחינת latency).
- לדוגמה: page fault יהיה ארוך יותר (יעתיק דף גדול יותר מהdisk למטען הדפים).

שאלה: דפים גדולים מקטינים את מספר ה-page faults. האם זהו יתרון? תשובה: לא, זהו אינו יתרון. מרבית היישומים אינם רגשים לתוכה של ה-page fault כלל. חלק מהיישומים, למשל web services, רגשים להשיה המקסימלית שיכל לגרום page fault בודד, ולכן עבורם דפים גדולים עלולים להזיק (כי דפים גדולים אכן מקטינים את מספר ה-page faults, אבל גם מאריכים את זמן הביצוע של כל page fault).

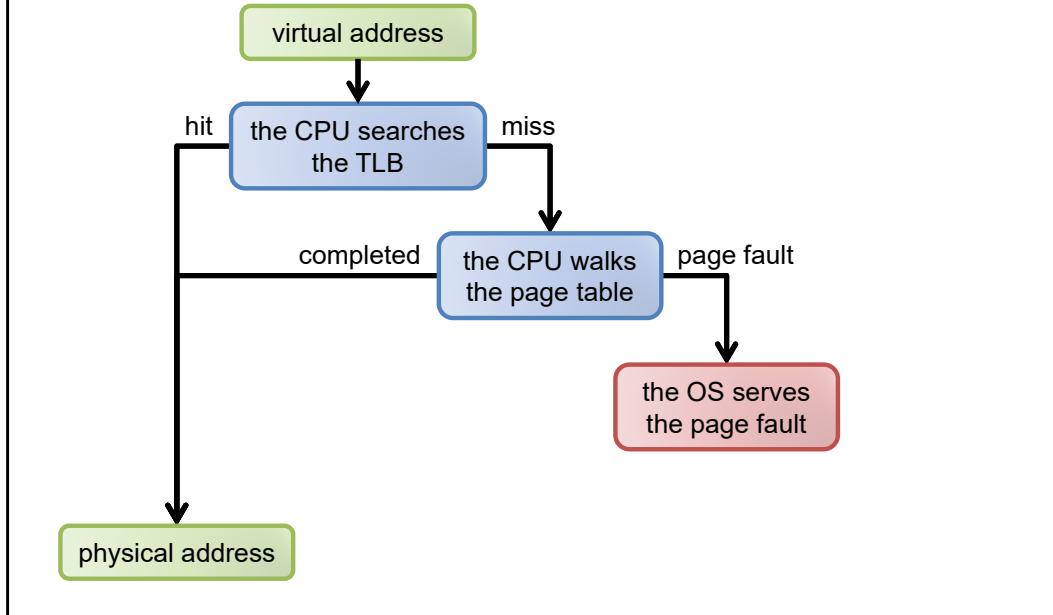
שאלה: האם דפים גדולים עדיפים/cgiשה סדרתית לדיסק, כי הם מביאים את המידע למטען הדפים בפחות page faults ?

תשובה: לא, אין עדיפות לדפים גדולים כאשר הגישה לדיסק היא סדרתית. גרעין לינוקס משתמש במנגןן read-ahead אשר מזהה גישה סדרתית לדיסק ומביא מראש עוד מידע כדי לחסוך page faults .

תרגול 11

מבנה נתונים בגרעין לניהול זיכרון
מנגןן copy-on-write
מנגןן demand paging

סיכון השיעור שער



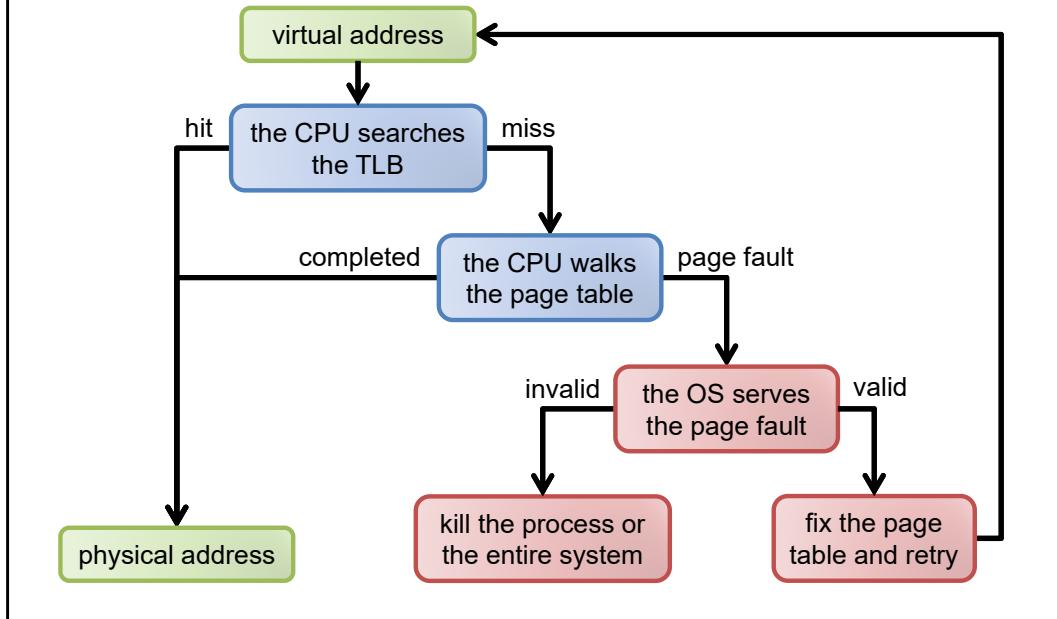
השלבים בכחול מתבצעים ע"י החומרה, השלבים באדום ע"י מערכת הפעלה.

יתכן מצב נוסף שאינו מופיע בשקף: חיפוש ב-TLB שMOVIL ישירות לחריגת דף, כלומר בלי page walk באמצעותו.

זה יכול לקרות אם למשל קוד משתמש מנסה לכתוב לכתובת וירטואלית שנמצאת ב-TLB, אבל ההרשאות המתאימות לכתובת זו לא מאפשרות כתיבה.

במצב זה המעבד ייצור חריגת דף למוראות שלא הייתה "החטאה" ב-TLB.

מה נלמד היום?



תזכורת לגבי חריגות (exceptions) מסוג **page fault** : בסיום הטיפול בחיריגה כמו **page fault** המעבד יחזור לבצע את הפקודה שגרמה לחריגת.

TL;DR

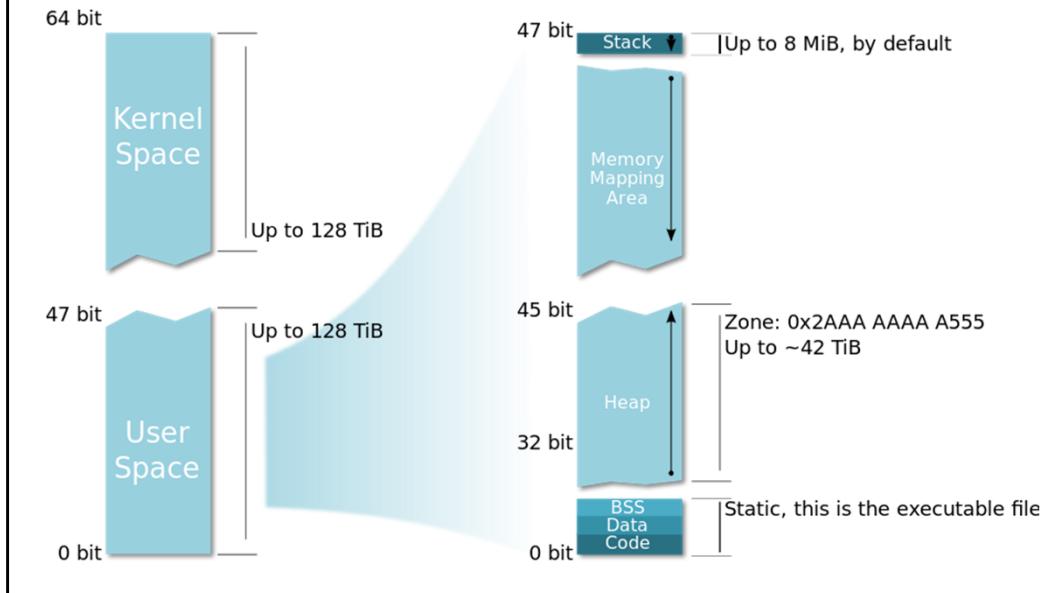
- זיכרון וירטואלי הוא **מנגנון משולב** חומרה-תוכנה:
- המעבד מתרגם את הכתובת הווירטואלית לכתובת פיזית בעת גישה לזכרון באמצעות הליכה בטבלת הדפים (page table walk) – ראיינו בתרגול הקודם הקודם.
- מערכת הפעלה מגדירה את טבלת הדפים, וכן מגדירה את המיפוי בין כתובות וירטואליות לפיזיות – את זה נראה היום.
- לינוקס מנהלת את טבלאות הדפים של תהליכים בצורה "עצלה/דוחינית" (lazy) כדי לחסוך זיכרון וזמן מעבד.
- לינוקס דוחה ככל הנימן העתקת זיכרון מהאב לבן באמצעות copy-on-write.
- לינוקס מקצה מסגרות פיזיות לתהליך בצורה עצלה ע"י demand paging.

העקרונות שנציג בתרגול זהו כלליים ולא תקפים רק לلينוקס, כי רוב מערכות הפעלה משתמשות במנגנונים עצלים כדי לנוהל זיכרון.
כמו כן, יש עוד מנגנונים עצלים בלינוקס שלא נלמד עליהם בקורס.

מבנה נתונים בגרעין לניהול זיכרון

טבלאות דפים, מרחבי זיכרון, אזורים זיכרון, ...

מרקם הזיכרון של תהלייר



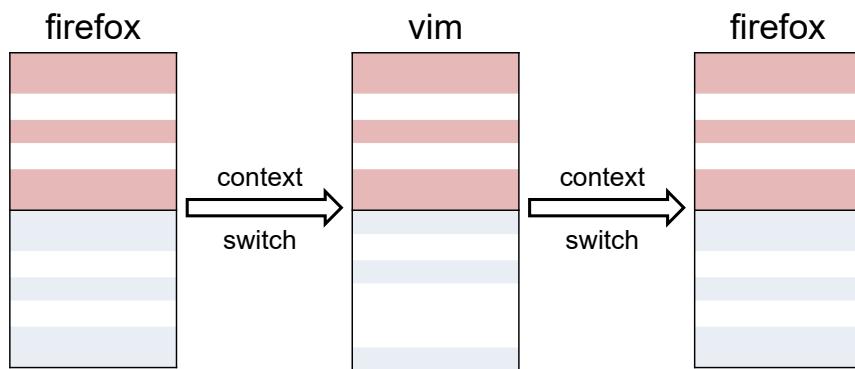
Picture from: https://en.wikibooks.org/wiki/The_Linux_Kernel/Memory

מרחב הזיכרון של תהלייר

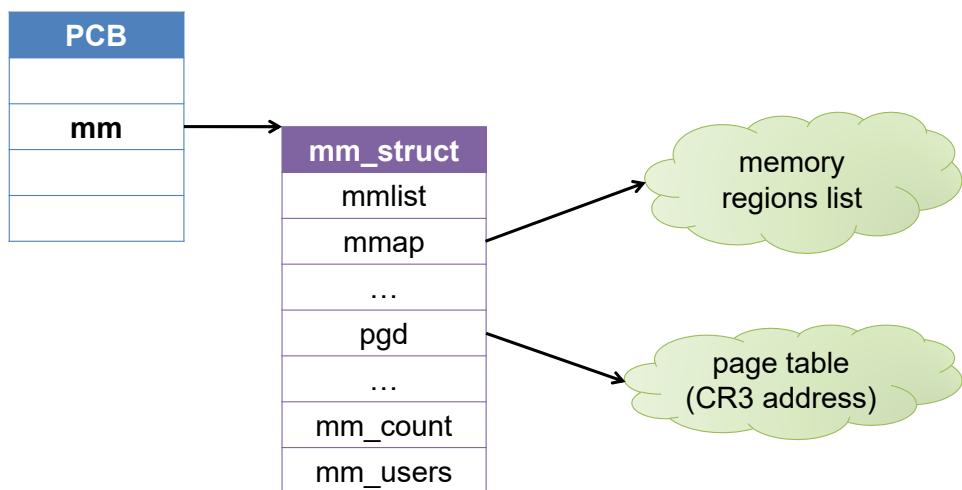
- במעבדי $64x$ רוחב כתובת וירטואלית הוא 48 ביט.
- ↪ גודל מרחב הזיכרון הווירטואלי של כל תהלייר הוא: $B = 256 \text{ TB} = 2^{48} \text{ B}$.
- בlianoks, מרחב הזיכרון הנ"ל מחולק לשניים:
- **128TB העריניים** – מרחב הזיכרון של הגרעין.
 - במרחב זה נשמרים כל מבני הנתונים והקוד של מערכת הפעלה, ובפרט: תוכי הריצה (queues), מחסניות הגרעין, כל טבלאות הדפים כל התהליכיים השונים, וכו'.
 - מרחב הגרעין לעולם אינו מפונה לדיסק (לעולם אינו swapped).
- **128TB התחתוניים** – מרחב הזיכרון של המשתמש.
 - במרחב זה נשמרים קוד התוכנית, המחסנית, הערימה, ואזורי זיכרון נוספים.

מרחב הגרעין

- מרחב הגרעין **משותף לכל התהליכים** כי הוא ממוקם לאותו מיקטן בזיכרון הוירטואלי של כל התהליכים.
- באופן זה, הכתובת (הוירטואלית) של כל אובייקט בגרעין נשארת קבועה בכל מרחב הזיכרון של כל התהליכים.



מפת רחוב הזיכרון של תהליך



מתאר הזיכרון של תהלייר

- השדה `mm` של כל PCB מצבייע אל **מתאר מרחב הזיכרון** (memory descriptor) של אותו תהלייר.
- מתאר מרחב הזיכרון מיוצג ע"י מבנה מסווג **.mm_struct**.
- תהליכיים אשר חולקים אותו מרחב זיכרון, כמו חוטים למשל, מצבייעים על אותו מתאר מרחב זיכרון.
- ערך שדה `mm` של חוט גרעין הוא `NULL`.
- חוט גרעין** הוא תהלייר שנוצר ע"י מערכת הפעלה כדי להריץ מושימה כלשהי של מערכת הפעלה.
 - לכל חוט גרעין יש PCB משלהו, דומה לתהלייר רגיל.
 - חותי גרעין לדוגמה: `ksoftirqd`, `kswapd`, `khugepaged`.

מודדר בקובץ `grub.h` include/linux/sched.h ..

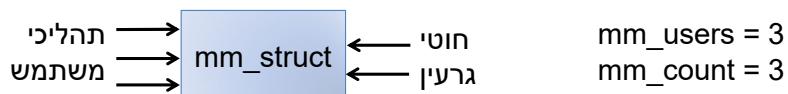
חוטי גרעין (kernel threads)

- מבחרנית זיכרון, חוט גרעין ניגש אך ורק למרחב הגרעין (128TB
 - העלויונים אשר משותפים לכל התהליכים במערכת).
- לחוט גרעין אין שם רכיב במרחב המשתמש, למשל, אין לו עירימה.
- חוט גרעין רץ רק בהרשאות גרעין ($CPL = 0$) על מחסנית הגרעין שלו.
- لكن בלינוקס חוט גרעין לא מקבל מרחב זיכרון עצמו, אלא פשוט פועל במרחב הזיכרון של תהליך המשתמש שרצץ לפניו.
- כאשר מערכת הפעלה מחליפה הקשר מתהליך רגיל לחוט גרעין, היא לא מחליפה את מרחב הזיכרון.
- במקרים פשוטות: גיסטר CR3 נותר ללא שינוי כדי שיצביע על טבלת הדפים של התהליך הרגיל.

הערה: חוטי גרעין לא נוצרים ע"י קריאות המערכת (`clone()` / `fork()`).

שדות במתאר מרחב הזיכרון של תהליך

- **mm_users** – כמה תהליכי משתמש חולקים את מרחב הזיכרון?
- **mm_count** – כמה תהליכי משתמש + חוטי גרעין חולקים את מרחב הזיכרון? כל תהליכי המשתמש ייחד נחשבים כאחד, אבל כל חוט גרעין נספר בנפרד.



- כאשר $0 == \text{mm_users}$, מערכת הפעלה מפנה את כל אזורי הזיכרון של המשתמש (מחסנית, ערים, קוד, וכו').
- כאשר $0 == \text{mm_count}$, מערכת הפעלה מפנה את מתאר מרחב הזיכרון כולו (וטבלת הדפים כולה).
- **mm_mm_count** מונע פנוי מרחב זיכרון כאשר הוא בשימוש ע"י תהליך גרעין.

Example scenario: process A **exits** and is switched by a kernel thread.

This thread uses the memory domain of the last process in the CPU - so we still cannot free it.

שדות במתאר מרחבי הזיכרון של תהלייר

- **mm_list:** קישור לרשימה הגלובלית של מתארים מרחבי הזיכרון (מטיפוס `list_head`), הנחלה ע"י כל התהלייכים.

- **pgd:** הכתובת של שורש טבלת הדפים.
- זה הערך שייטע ל-`CR3` כאשר התהלייר יזוםן ליריצה.

מכיל כתובות פיזית או וירטואלית?

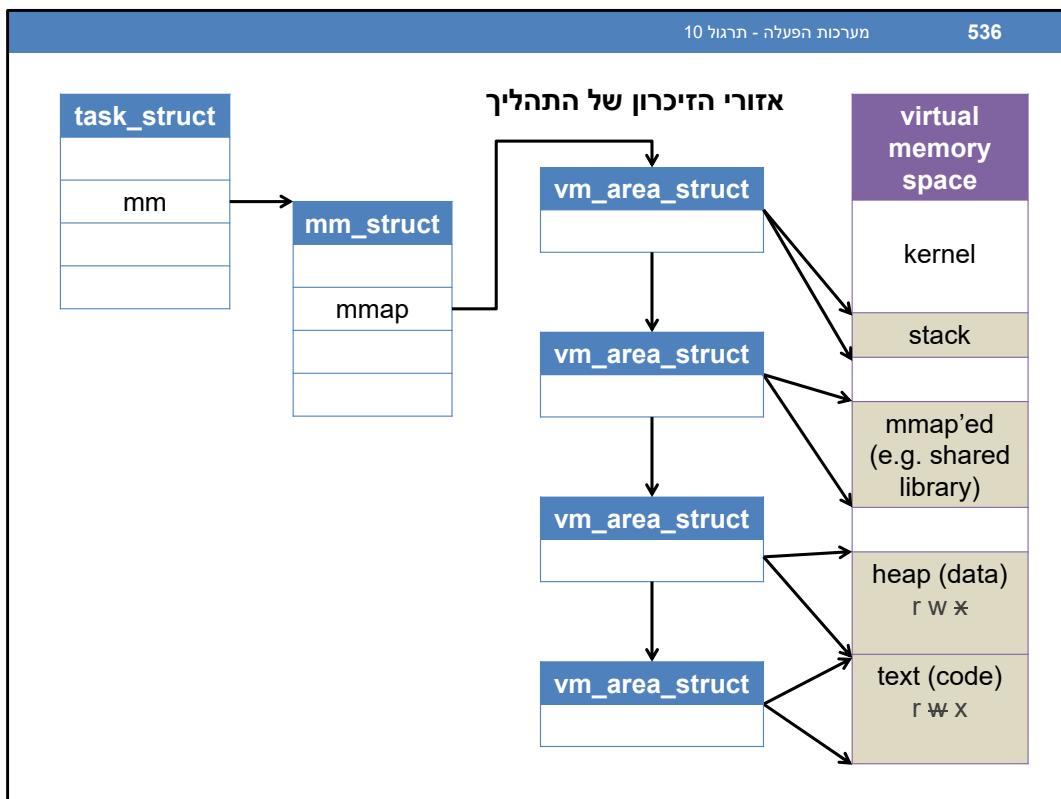
- **mm_map:** רשימה ממוינת של אזורי הזיכרון – נראה בהמשך.

- **rss:** מספר המסגרות בשימוש (דפים מגובים בזיכרון).

- **total_vm:** מספר דפים כולל באזורי הזיכרון.

האם `vm rss`, `total_vm` יכולים להיות שונים?

1. כתובות פיזית. רגיסטר `CR3` חייב להציגו לכתובת פיזית, אחרת תהלייר תרגום הכתובות (`page walk`) ייכנס לרקע אינסופית.
2. כן, מפני שניתן להיות דפים בזיכרון הווירטואלי שאינם ממוקמים במסגרות בזיכרון הפיזי. למשל במקרה `demand paging`.
בכל מקרה תמיד $\text{mtk}'s \text{ vm_rss} \leq \text{total_vm}$.





אזור זיכרון

- הגרעין מנהל את מרחב הזיכרון של התהילר ע"י חלוקה **לאזור זיכרון** (virtual memory regions).
- אזור זיכרון הוא רצף כתובות במרחב הזיכרון של התהילר אשר שייר לתחום של 128TB התחתיונים (כלומר לא לגרעין).
 - .1 אזור זיכרון אינם חופפים.
 - .2 לכל אזור הרשות קריאה/כתיבה/ביצוע משלו.
 - .3 תהילר יכול לגשת רק לכתובת שנמצאת באזור זיכרון כלשהו.
 - .4 כתובות התחלתית וגדל של אזור זיכרון הם כפולות של גודל הדף.
 - .5 זכרו שיחידת ההקציה הבסיסית של מבנה הזיכרון הוירטואלי היא דף.

הקצתת אזור זיכרון לתהילר לא מוסיפה מיד דפים ולא מעדכנת את טבלת הדפים---הההקציה נדחית עד לרגע בו הם נדרשים, כפי שנראה בהמשך.

מתאר אזור זיכרון

- אזור זיכרון מאופיין ע"י **מתאר אזור זיכרון**, שהוא רשומה מטיפוס [.vm_area_struct](#).
- שדות במתאר אזור זיכרון:
 - `vm_start`: כתובת התחלת של אזור הזיכרון.
 - `vm_end`: כתובת אחת אחרי האחרונה של אזור הזיכרון.
 - `vm_next`: מצביע למתאר אזור הזיכרון הבא ברישימה המקשורה של האזוריים.
 - `vm_mm`: מצביע חזרה למתאר מרחב הזיכרון המכיל את האזור.
 - `vm_flags`: דגלים המציינים תכונות של האזור.
 - `vm_page_prot`: ערכי ביטים שונים שיוצבו לכל הכניסות של טבלת הדפים עבור הדפים באזור.

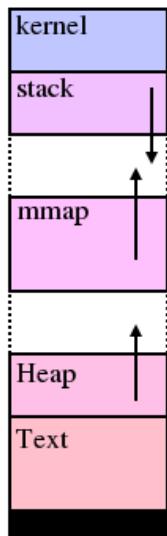
מוגדרת בקובץ גרעין [. include/linux/mm.h](#)



הרשאות של איזור זיכרון

- הדגלים המציינים את הרשאות האיזור נשמרים בשדה **vm_flags** והם מאפשרים לארען לסוג גישות חוקיות ולא חוקיות לדפים באיזור.
- VM_READ, VM_WRITE, VM_EXEC – האם מותר לקרוא/לכתוב/לבצע נתונים בדף באיזור.
- VM_MAYREAD, VM_MAYWRITE, VM_MAYEXEC – "הרשאת הרשאה" לכל אחת מההרשאות הנ"ל.
 - לדוגמה VM_MAYWRITE קובע האם מותר להדליק את VM_WRITE.
- הדגלים האלה קשורים לקריאת המערכת (`mprotect()` – מעבר לחומר הקורט).
- VM_SHARED – האם צריך לשתף דפים באיזור זה עם תהליכי בן.
- VM_LOCKED – אסור לפנות את הדפים באיזור מהזיכרון לדיסק.

מתי נוצרים אזורי זיכרון?



- עם היוזר, תהליך מקבל מרחב זיכרון עם אזורי זיכרון הבאים:
 - .1 אזור לקוד (code או text).
 - .2 אזור לנוטונים סטטיים (data).
 - .3 אזור לעירמה של הזיכרון הדינמי (heap).
 - .4 אזור למחסנית user mode.
 - .5 אזורים נוספים: אחד לפרמטרים של שורת הפקודה, אחד למשתני מערכת.
- הוספה של אזוריים נוספים מתאפשרת באמצעות קריית המערכת (`mmap()`).

ניהול זיכרון דינמי

- תהיליך משתמש יכול להקצות או לשחרר זיכרון באמצעות פונקציות הספרייה `malloc()`, `free()`.
- `malloc()` מ Chapman בлок זיכרון פנוי באזור הערימה.
- במידה ואין מספיק זיכרון פנוי בערימה, `malloc()` פונה לקריאה המערכת `brk()` כדי להגדיל את איזור זיכרון הערימה.



Today, `malloc()` calls `sbrk()` for small allocations below `MMAP_THRESHOLD` and `mmap()` for larger allocation requests.

The reason we have both `sbrk()` and `mmap()` system calls is historical: `sbrk()` was the first, and `mmap()` was introduced later.



קריאה המערכת (sbrk)

```
void *sbrk(intptr_t increment);
```

- פועלה: מגדילה או מקטינה את הקצה העליון (program break) של אזור הזיכרון של הערימה (heap).
 - אם > 0 , אז sbrk מגדיל זיכרון נוסף בערימה.
 - אם < 0 , אז sbrk משחררת זיכרון מהערימה.
 - אם $= 0$, אז sbrk מוחזירה את ראש הערימה הנוכחי.



קריאה המערכת (`mmap()`)

```
void *mmap(void *addr, size_t length,  
           int prot, int flags,  
           int fd, off_t offset);
```

- פועלה: יוצרת איזור זיכרון חדש ומוסיףו אותו לרשימת איזורי הזיכרון של התהילה.
 - .1 אם `fd` הוא אינדקס של קובץ פתוח, אז האיזור החדש ימפה את אותו קובץ.
 - .2 אם `-1 == fd`, אז האיזור החדש הוא אונוני.

בשביל להגדיר איזור אונוני צריך להעביר גם דגל `MAP_ANONYMOUS`.

סיווג אזרוי זיכרון

anonימי

(anonymous)

- אזרור הזיכרון מכיל מידע שאינו הקשור לשום קובץ, אלא לזיכרון הדינמי של התהילה.
- במידה וחסר זיכרון במערכת, הגሩין יכול לפנות דפים אונונימיים למחיצה מיוחדת swap – area swap.

מה הסיווג של אזרוי הזיכרון:
ערימה, מחסנית, קוד ?

מוגבה קובץ

(file-backed)

- אזרור הזיכרון מכיל מידע שמקורו בקובץ.
- זכרו כי "קובץ" אינו בהכרח קובץ "רגיל" המאוחסן בדיסק.
- הקובץ נקרא " ממופה לזכרון ", או memory mapped file
- קריאה/כתיבה לאזרור הזיכרון מתורגמת לקריאה/כתיבה למקום המתאים בתוך הקובץ.

תשובה: המחסנית והערימה – אזרוי זיכרון אונונימיים.
 הקוד – אזרוי זיכרון מוגבה קובץ, שכן הוא מגבה את הקובץ הבינארי של התוכנית.
 לא ניתן לכתוב לאזרור זה (אין הרשותות כתיבה במתאר האזרור ובטבלת הדפים), ולכן לעולם אינם dirty. דהיינו, לא נדרש לכתוב אותו חזרה לדיסק עם סיום התהילה.

מרחבי זיכרון וקריאות מערכת

- חוטים הנוצרים ע"י קריית המערכת (`clone`) משתפים את מרחב הזיכרון ע"י הצבעה לאותו מתאר מרחב הזיכרון של תהיליך האב.
- יש להגדיל את מונח השיטוף (`mm`) של מתאר מרחב הזיכרון של תהיליך האב.
- קריית המערכת (`execv`) ודומותיה טענות תהיליך חדש ולכן הן משחררות את מרחב הזיכרון ומ Katzot אחד חדש.
- קריית המערכת (`fork`) מקצתה ל תהיליך הבן מרחב זיכרון משלה.
- במקרה שכזה צריך להעתיק את מרחב הזיכרון של האב לזה של הבן.
- בפועל, בדרך כלל אין באמת העתקה בזכותו מגנון **copy-on-write**.

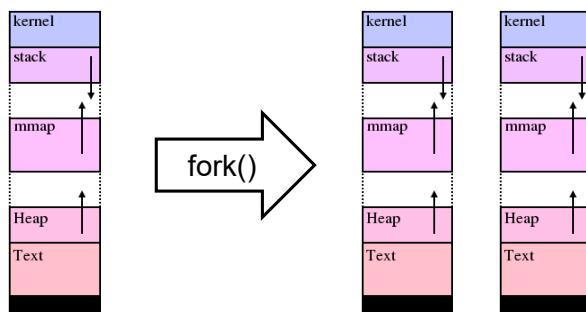
הפוך



COPY-ON-WRITE מנגנון

מווטיבציה למנגנון COW

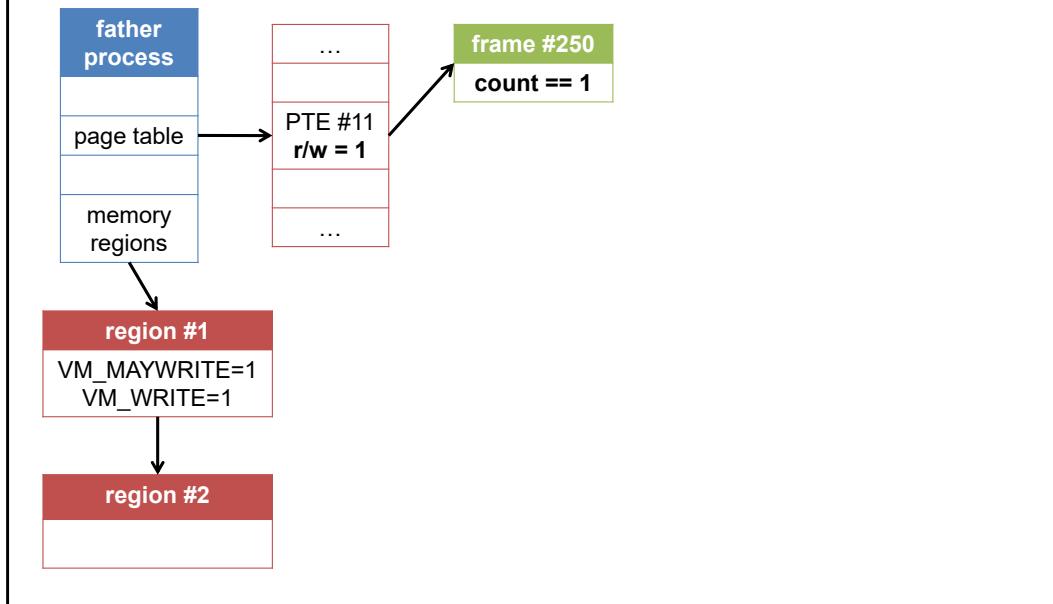
- קריית המערכת (`fork()`) דורשת להעתיק את מרחב הזיכרון של האב לשנה של הבן. אבל העתקה פשוטה של מרחב זיכרון היא:
 - .1. איטית: הרבה זמן נדרש להעתיק כל הדפים.
 - .2. אולי מיותרת: מרחב הזיכרון של תהליך הבן ימחק אם הבן יטען תוכנית חדשה ע"י קרייה ל-`(exec()` מיד בתחילת ריצתו.



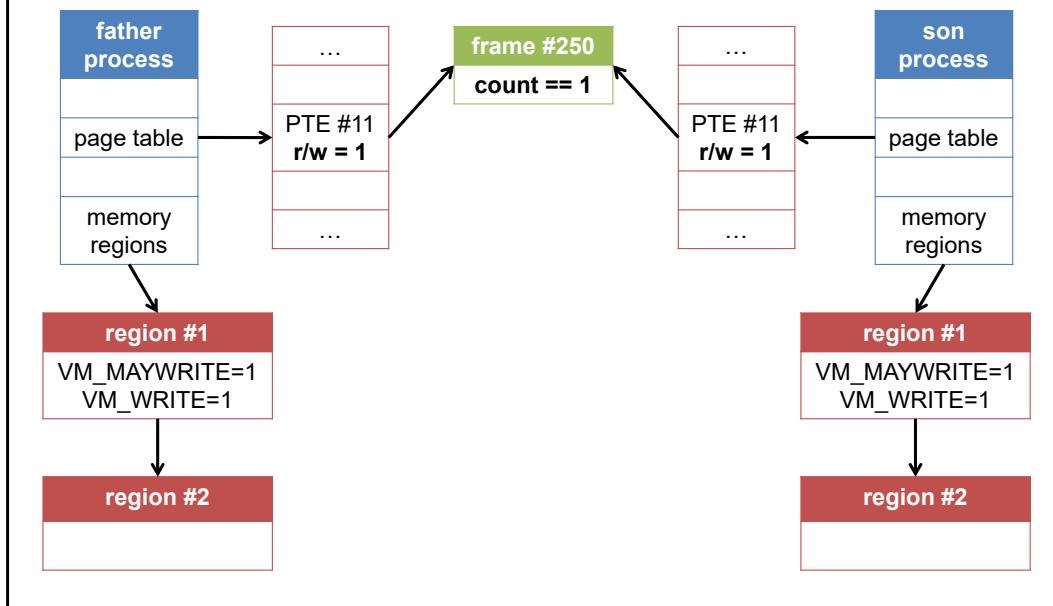
הפתרון: copy-on-write (COW)

- הרעיון של מנגנון (COW) copy-on-write הוא:
 - דף הנitinim לכתיבה שאינו יכול להיות משותפים (לדוגמה, המחסנית), מוגדרים בתחילת המשותפים אבל **后备יקים לעותק פרטי** כאשר אחד **התהילכים השותפים** (האב או הבן) מנסה לכתוב אליהם לראשונה.
 - שאר הדפים (כדוגמת דפי קוד או דפי נתונים לקריאה בלבד) הופכים לשותפים בין מרחביו הדינמי של האב והבן.
- מנגנון COW פותר את שתי הבעיה שהוצעו קודם:
 - COW מקטין את זמן הביצוע של (`fork` כי הוא "פורט לתשלומים" את העתקה של כל מרחב הזיכרון להרבה העתקות קטנות בגודל דף שיתבצעו בעתיד---בכל כתיבה ראשונה לדף שאינו משותף.
 - במידה ותהליך הבן יבצע מיד (`execv`), מרחב הזיכרון שלו יימחק וכך תיחסר רוב פעולות העתקה.

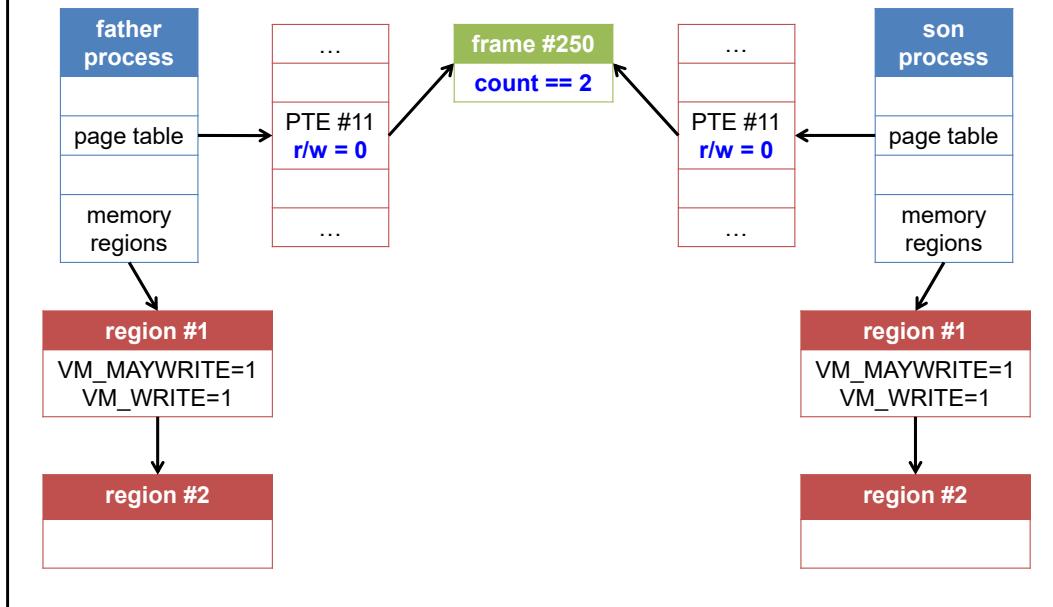
דוגמה: לפני קראת מערכת (fork())



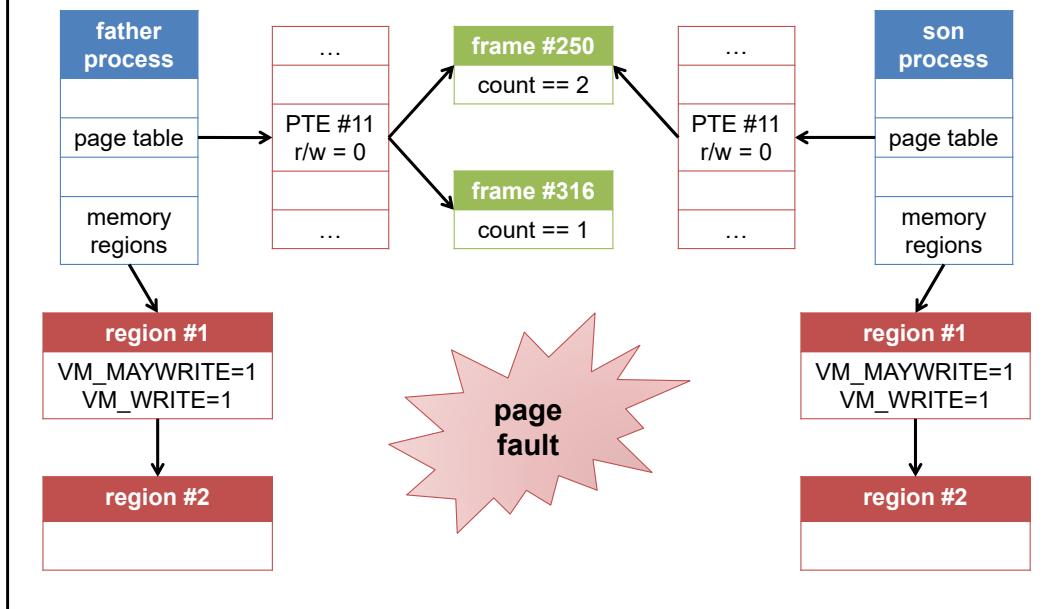
דוגמה: אחריו קריית מערכת fork()



דוגמה: אחריו קריית מערכת fork()

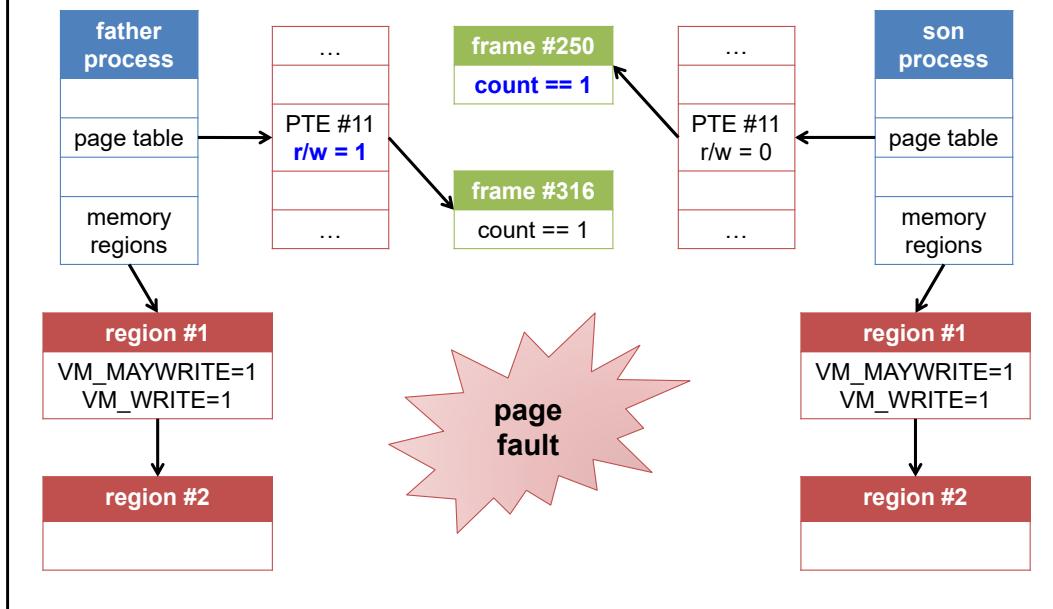


דוגמה: תהליך ראשון מנסה לכתוב



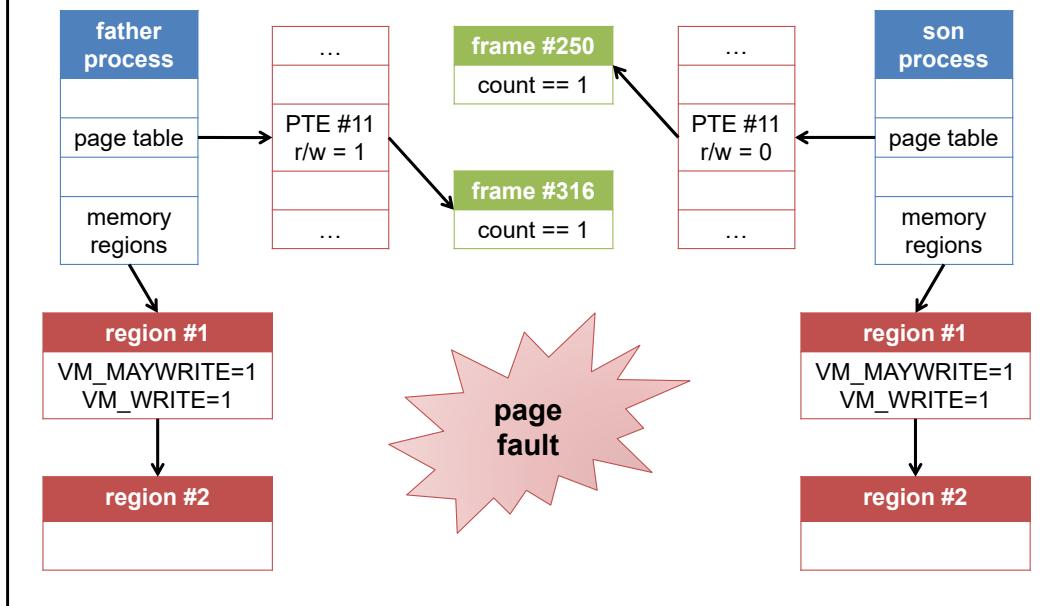
התהליך הראשון שמנסה לכתוב---לא משנה אם זה האב או הבן---הוא זה שיקבל חריגת דף
ואז יעתיק את המסגרת.

דוגמה: תהליך ראשון מנסה לכתוב

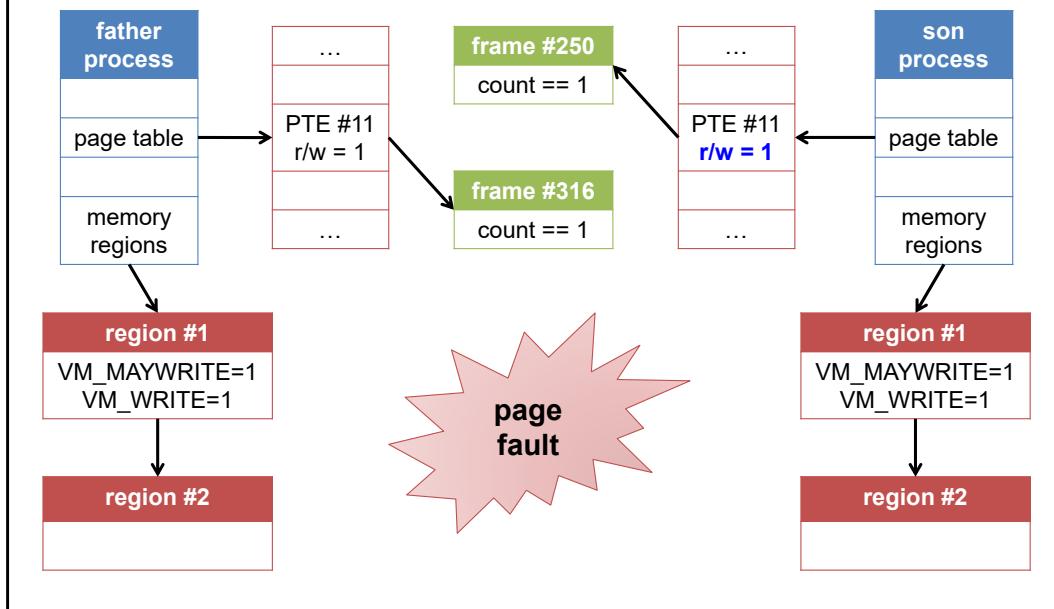


התהליך הראשון שמנסה לכתוב---לא משנה אם זה האב או הבן---הוא זה שיקבל חריגת דף
ואז יעתיק את המסגרת.

דוגמה: תהליך שני מנסה לכתוב



דוגמה: תהליך שני מנסה לכתוב



COW הוא דוגמה למנגנון "עצל"

- המנגנון מורכב משני שלבים:
 - .1. הגנה על דפים במסגרת קריית המערכת (fork).
 - .2. שכפול מסגרות לאחר נסיוון כתיבה שגרם ל- page_fault.
- שלב 1 (הגנה על דפים) מאפשר לארען לדוחות את שלב 2 (שכפול מסגרות) ככל הנitin.
- **שאלה:** האם בהכרח שיפרנו את הביצועים בעזרת COW?
 - אם, בסופו של דבר, האב והבן יכתבו לכל הדפים שלהם, לא חסכנו שום עבודה והיינו יכולים להעתיק את כל מרחיב הזיכרון מלכתחילה.
 - למעשה אפילו הוספנו עבודה מיותרת של עדכון טבלאות הדפים בשלב 1 + תקורה נוספת של חריגות דף בשלב 2.
- אבל באופן השימוש הנפוץ (fork+execv) חסכנו הרבה עבודה.
- כי שילמנו רק על שלב 1 שהוא זול יותר מאשר שלב 2.

COW: העתקת מרחב זיכרון לתהיליך בן

- הfonקציית `(mm_copy, המופעלת מתוך (do_fork, "מעתיקה" את מרחב הזיכרון של תהיליך האב לתהיליך הבן.`
- `לכל איזור זיכרון של האב:`
 - `מעתיקה את מתאר איזור הזיכרון לתהיליך הבן (עותק חדש ונפרד).`
 - `מעתיקה את הכניסות המתאימות מטבלת הדפים של האב לזו של הבן.`
 - `לכל דף באיזור הזיכרון, מגדילה את מונח השיתוף של המסגרת המתאימה.`
- `טהיליך ההגנה: קריית המערכת fork ניגשת לדפים:`
 - `שאים משותפים (VM_SHARED כבוי)`
 - `שניתן לאפשר בהם כתיבה (VM_MAYWRITE VM_DLOCK)`
 - `ומכבה את הביט W/z ב-PTE של אותו דף.`

COW: טיפול ב-page fault

תרחיש הטיפול:

- האב או הבן מנוטים לכתוב לדף מוגן ע"י COW.
- המעבד ניגש לסייעות הבקשה ב-PTE של הדף, ומגלה כי W/z קבוע.
- המעבד יוצר חריגת דף (page fault).
- הגרעין מטפל בחריגת, ובודק שהדף **שייך** לאחד מהאזורים הזיכרון והגהישה בכלל חוקיות (דגל E_WRITE VM_DLOCK במתאר האזור).
- הגרעין בודק את ערך המונחה השיתוף של המסגרת:
 - אם $1 > \text{count}$, מזמנים מסגרת חדשה, מעתיקים אליה את המסגרת המקורית, ומצביעים את הדף למסגרת החדשה.
 - במסגרת החדשה מבוצע --count .
 - במסגרת החדשה מוצב $\text{count} = 1$.
 - בעותק החדש מאופסרת הכתיבה.
- אחרת ($1 == \text{count}$), הגרעין פשוט מאפשר כתיבה בדף ע"י הדלקת הדגל W/r.

מנגנון DEMAND PAGING

דוגמה קידמית

```
char *a = (char*)mmap(NULL, 4096,  
PROT_READ | PROT_WRITE, MAP_ANONYMOUS,  
-1, 0); // OS doesn't allocate memory,  
// only updates the memory region list  
  
x = a[0]; // page fault  
// OS maps the page to the zero page  
  
a[0] = 6; // another page fault  
// OS allocates a new frame  
// and copies the zero page into it
```

הקצת מסגרות לפי דרישת (Demand Paging)

- כasher tahlir mukzah zikron bamezuot kriyat ha-sistema (mmap), linokh mukzah masgerot lifi drisha (demand paging).
 - .1. בשלב הראשון, רק רשיימת אזורי הזיכרון מתחדנת.
 - .2. הכניסות המתאימות בטבלת הדפים עדין לא מצביעות למסגרות ("סימון בית 0 == present").
 - .3. המסגרת מוקזית או מועתקת מהdisk רק בניסיון הגישה הראשון לדף, בעקבות page fault.

סוגי חריגת דף

Major page fault

- המידע הנדרש נמצא על הדיסק ולא בזיכרון.
- דף אונוניי בswap
- לכן הגישה בהכרח חוסמת

Minor page fault

- המידע הנדרש, כשלעצמם איןנו דורש גישה לדיסק CoW
- הקצתת זיכרון אונוניי
- קריאה דף שנמצא כבר במתomon הדפים
- אבל תתקן בכל זאת גישה לדיסק
- אם צריך לפנות מקום בזיכרון בעבור הדף החדש

האם minor page fault יכול לדרש גישה לדיסק ולגרור יציאה להמתנה? כן. לרוב מיינור איננו דורש גישה לדיסק מכיון שהמידע שורצים לקרוא או נכתבו כבר בזיכרון וכן רק לתקן את המיפוי. אבל קיים מקרה קצה בו המידע נמצא בזיכרון אבל אין מספיק זיכרון כדי לספק את הבקשה, במקרה זה יבוצע swap out לדף אחר לדיסק כפי לפנות זיכרון למידע שלנו (העתקה של קובץ בWOW).

מה כולל WOW? כתיבה לדף במתomon הדפים, כתיבה לדף האפסים, כתיבה לדף שעבר fork. בכל מקרים אלו יתבצע נסيون כתיבה (חוק) למסגרות אשו מוגנות מפני כתיבה -> יבוצע שיכפוף של המסגרת ותיקון של הצבעת הדף למסגרת החדשה.

טיפול בחריגת דף – Demand Paging

תחילה, הגען בודק אם הכתובת שג儒家 להריגה היא חוקית.

במידה וכן:

.1. אם איזור הזיכרון מפנה קובץ שנמצא בדיסק, יש לטען את המסגרת מהdisk (major page fault).

.2. אם מעולם לא ניגשו לאיזור הזיכרון (זיכרון אונומי קר):

.1. אם הגישה לכתיבה, מוקצת מסגרת חדשה מלאה באפסים (minor page fault).

.2. אם הגישה לקריאה, הכניסה בטבלת הדפים מצביעה על מסגרת של דף קבוע מיוחד ממולא אפסים, הקורי ZERO_PAGE.

• דף זה מסומן read-only, כך שבכתייה הראשונה לדף הוא ישוכפל לעותק פרטיל פי. שיטת COW.

• בכל המקרים, הקצת מסגרת חדשה עשויה לדרוש גם הוספה כניסה מתאימה בכל הרמות של טבלת הדפים.



חריגת דף (page fault)

- החומרה מתריעה באמצעות **חריגת דף** על:
 - גישה לדף שאינו נמצא בזיכרון, כלומר הביט `present==0` בכתובת הדפים.
 - גישה לא חוקית (שלא לפי ההרשאות בטבלת הדפים) לדף שנמצא בזיכרון, למשל ניסיון כתיבה לדף שモותר לקרוא בלבד.
- חריגת דף מפעילה את שגרת הטיפול המוממשת בפונקציית הגרעין `.do_page_fault()`.
- בסיום הטיפול בחיריגת **מבוצעת מחדש** ההוראה שהרמה לה.
 - אלא אם כן, כמובן, הטיפול בחיריגת הורג את התהילר.

מודדרת בקובץ הגרעין `.arch/i386/mm/fault.c`

לא כל חריגת דף היא תקלת!

- לינוקס נדרש לנתח את נסיבות החריגה ולהחליט אם היא חוקית וכיitzד לטפל בה.
- כתיבה לדף שמותר לקרוא בלבד עשויה להיות חוקית, למשל ב-WOW.
- קריאה מדף שעבר ל-swap היא חוקית; הגሩין צריך להחזיר את הדף לזיכרון.
- כדי שמערכת הפעלה תוכל לטפל בחറיגת הדף, החומרה מעבירה לשגרת הטיפול קוד שגיאה של 3 ביטים נשמר במחסנית:
 - **ביט 0** כבוי: גישה לדף שאינו בזיכרון ($0 == \text{present}$). אחרת, גישה לא חוקית לדף בזיכרון.
 - **ביט 1** כבוי: הגישה הייתה לקריאה או לביצוע קוד. אחרת, הגישה הייתה לכתיבה.
 - **ביט 2** כבוי: הגישה כהמעבד ב-kernel-mode. אחרת, הגישה ב-user-mode.
- כמו כן, החומרה מעבירה את הכתובת הווירטואלית שגרמה לחריגה ברגיסטר CR2.



טיפול בתקלות

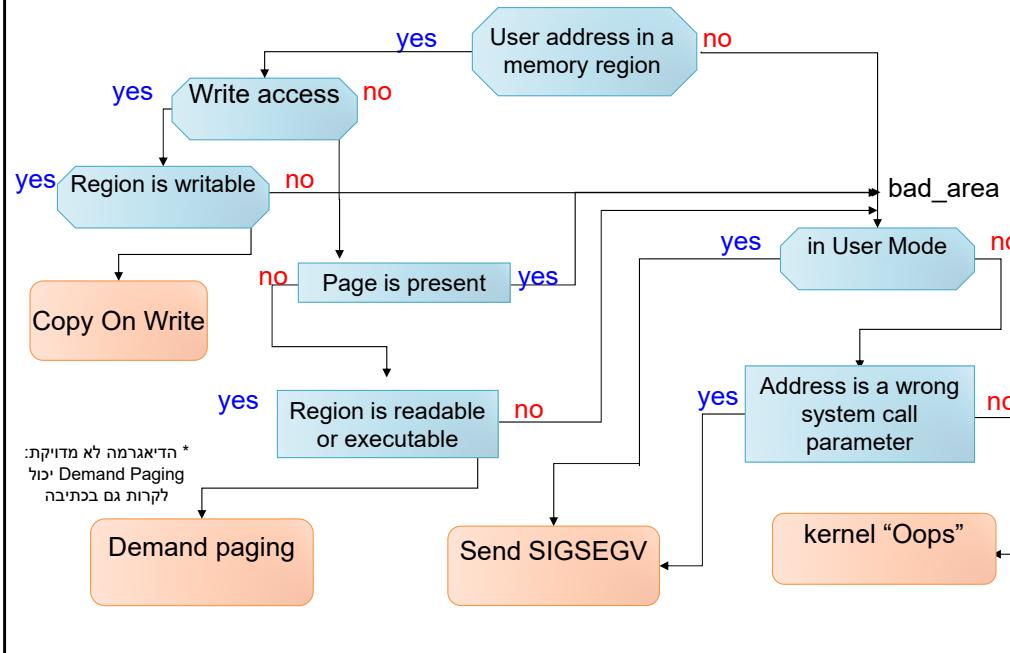
- החריגת מסווגת כתקלה (גישה לא חוקית) אם:
 - .1. הפעולה (קריאה או כתיבה) לא מורשית לפי הרשות האזר.
 - .2. גישה מקוד משמש לדפי הגערין.
 - .3. גישה לכתובת בתחום המשמש שאינה שייכת לשום אזור זיכרון.
- אם הגישה הייתה מקוד תהלייר משמש, נשלח לתהלייר signal SIGSEGV, לציין "גישה לא חוקית לזכרון".
- אם הגישה הייתה מקוד גרעין, מוכרצת תקלה מערכת – oops.

לכל 3 זה יש חריג אחד – כתיבה למחסנית, שעלולה "לגלוש" מעבר לאזור הזיכרון הנוכחי שלה.

פעולת כתיבה ייחודית למחסנית יכולה להגדיל אותה לכל היותר ב-32 בתים (פעולת push).

לכן, אם הפעולה היא כתיבה בהתאם להרשות, אזור הזיכרון הוא מחסנית (NVM_GROWDOWN דלוק), וכתובת הגישה היא עד 32 בתים מתחת לתחילת אזור המחסנית, מוקצת דף נוסף למחסנית וביצוע הכתיבה מאופサー.

סיכון: טיפול בחיריגת דף במצב משתמש



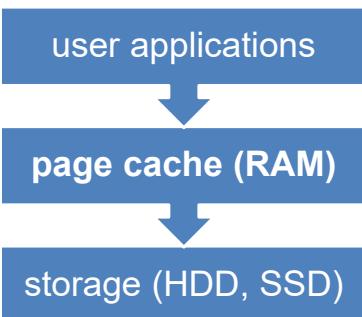
שימוש לב: הדיאגרמה לא מכילה את המקירה של הגדלת מחסנית!

תרגול 12

מטרון הדפים (Page Cache)
מבנה נתונים לניהול זיכרון פיזי
אופני גישה למטרון הדפים
אלגוריתם שחרור מסגרות (PFRA)

Special thanks to Gustavo Duartes who
provided many of the figures and illustrations

TL;DR



- התקני איחסון (לדוגמה דיסק קשיח או SSD) הם איטיים בכמה סדרי גודל מהזיכרון (RAM).
- תוכניות אשר ניגשות הרבה לדיסק עלולות לסייע מזמן ההשאה הארוכים.
- כדי להתגבר על הבעיה, מערכת הפעלה שומרת מידע מהדיסק בזיכרון, במבנה הנקרא מטמון הדפים.
- מטמון הדפים מצמצם את מספר הגישות לדיסק בזכות עיקרונו הлокליות (במרחב ובזמן).

локליות במרחב – גישה לכטבות סמוכות באותו הקובץ, למשל קריאה רציפה של קובץ גדול.
 לוקליות בזמן – גישה לאותה הכתובת בזמןים סמוכים, למשל תכנית אחת שעורכת קוד ותכנית שנייה שמהדרת (מקומפלט) אותו.

מטען הדפים

Page Cache

קצת היסטוריה: בגרסאות ישנות של לינוקס (לפני 2.4.10) הגרעין השתמש בשני מטמוניים:

- buffer cache השומר מידע מהדיסק ביחידות של בלוקים (512 בתים).
- page cache השומר מידע מהדיסק ביחידות של דפים.

למה צריך שני מטמוניים שונים? ציטוט מתוך UTLK2:

"Block I/O operations are most often used when the kernel reads or writes single blocks in a filesystem (for example, a block containing an inode or a superblock). Conversely, page I/O operations are used mainly for reading and writing files (both regular files and block device files), for accessing files through the memory mapping, and for swapping. Both kinds of I/O operations rely on the same functions to access a block device, but the kernel uses different algorithms and buffering techniques with them."

הבעיה: התקני איחסון איטיים

- התקני איחסון הם בעלי השהייה גבוהה יחסית לזכרון.
- זמני ההשהייה האופיינים (נכון לשנת 2020) בגישה אקרואית*:
 - זכרון (DRAM) – 100 ns
 - כונן SSD – 16 us
 - כונן HDD – 2 ms
- המרכיב הדומיננטי הוא זמן הzzת הראש הקורא (seek latency).
- *גישה אקרואית מוגדרת כקריאה/כתיבה של 8 ביביות כלשהי.

The numbers are taken from “Latency Numbers Every Programmer Should Know”: https://people.eecs.berkeley.edu/~rcs/research/interactive_latency.html

הפתרון: מטמון הדפים

- מטמון הדפים הוא אוסף מסגרות בזיכרון הפיזי השומרות עותק של קבצים שמקורם בהתקני אחסון.
- המסגרות של מטמון הדפים לא בהכרח רציפות בזיכרון הפיזי.

physical memory
page cache frame



כל גישה לדיסק עוברת דרך מטמון הדפים

a user process calls
read() / write()



the kernel looks up
the page cache (RAM)



the kernel must access the
storage device (HDD, SSD)

- כאשר תכנית מבקשת לקרוא/לכתחזק לדיסק, הגרעין בודק קודם אם המידע המבוקש נמצא במטמון הדפים.
- אם כן – הגרעין משרת את הבקשה ממטמון הדפים.
- אם לא – הגרעין יקרא את המידע מהדיסק ואז יוסיף אותו למטמון הדפים.
- בכל מקרה, הגרעין חייב להביא את המידע המבוקש ל זיכרון כי המעבד לא יכול לגשת שירות לדיסק.

עקרון הלוקאליות

локאליות במרחב

- אם המשמש ניגש לבית מסויים בדיסק, יש סיכוי גבוהה שהוא ייגש לבטים סמוכים בעtid הקרוב.
- לדוגמה: תוכנית הקוראת קובץ טקסט שורה אחר שורה.
- בשורה הראשונה – הגרעין יקרא את הדף הראשון (4KB) של הקובץ מהדיסק.
- בשורה השנייה – המידיע כבר קיים בזיכרון הדפים, ונקרא אותו ללא גישה לדיסק.

локאליות בזמן

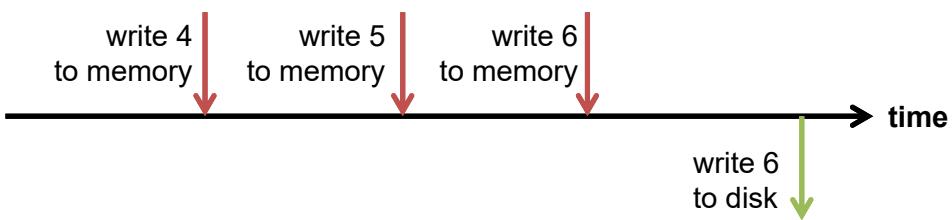
- אם המשמש ניגש בית מסויים בדיסק, יש סיכוי גבוהה שהוא ייגש שוב באותו בית בעtid הקרוב.
- לדוגמה: תהליך שעורך קוד ותהליך אחריו שמהדר אותו. התהליך השני יחסור גישה לדיסק אם הקובץ כבר קיים בזיכרון הדפים.

בדוגמה של התוכנית אשר קוראת שורה אחריה שורה, יש הנחה סטטוס שכל שורה מכילה כמה عشرות/מאות תוים ולכן היא בגודל כמה עשרות/מאות בתים – כלומר פחות מוגדל דף.

עקרון הלוקאליות הוא העומד גם מאחורי מטמוניים באופן כללי, למשל המטמוניים 3,2,1,1 של המעבד וכן ה-TLB שלמדו לנו.

כתיבה מושחת (write-back)

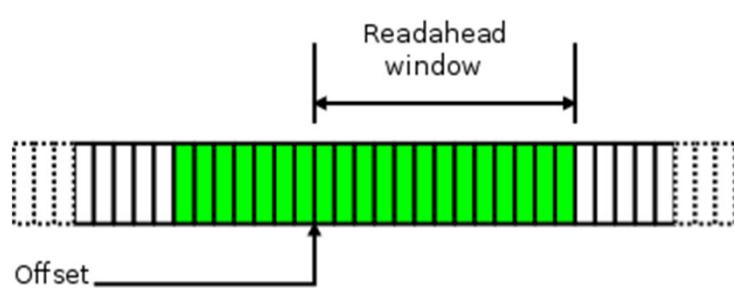
- כתיבות לדיסק נדחות כדי לחסוך (אול') כתיבות של ערכי ביןיהם.
- יתרון:** צמצום של מספר הכתובות לדיסק.
- שיפור ביצועים לא רק לתוכנית הכותבת אלא לכל המערכת: במידה והdisk מעומם בבקשת קריאה/כתיבה, אז כתיבה מושחת תפחית את מספר הכתובות הכלול (לעומת כתיבה מיידית, write-through).
- חיסרון:** אובדן אמינות - המידע בזיכרון עלול ללקת לאיבוד אם יש נפילת מתח.



לינוקס מבטיחה כי השינויים לעותק של הקובץ במתมอง הדפים יכתבו חזרה לדיסק רק בעת שחרור הזיכרון ע"י קריית המערכת (kunmap או ע"י קרייה מפורשת לקריאת המערכת sync().).

קריאה מראש (read-ahead)

- אם הגauważ חזזה גישה סדרתית לקובץ, הוא קורא למתਮן הדפים את המסגרות הבאות עוד לפני שהתהליך ניגש אליה.
- הקריאה מתרחשת ברקע, לא עוצרת את התקדמות התהליך, ולא מבזבצת כמעט זמן מעבד.
- יתרון:** פחתת החטאות במתמן הדפים במידה והחיזוי נכון.
- חיסרון:** בזיכרון זיכרון במידה והחיזוי שגוי.



חיסרון נוסף של קריאה מראש: במידה והדיסק עמוס בבקשת קריאה/כתיבה, אז העומס הנוסף של קריאה מראש עלול "לחנוק" את הדיסק ולפגוע בBITS של כלל המערכת.

The figure is taken from: <https://lwn.net/Articles/372384/>

מבנה נתונים לניהול זיכרון פיזי

סיווג מסגרות בזיכרון הפיזי

מסגרות אוניברסליות

- מכילות מידע שאינו הקשור לשום קובץ, אלא לזכרון הדינמי של התהיליך.
- לדוגמה: אישור הזיכרון של המחסנית והערימה.
- במידה וחסר זיכרון במערכת, הגሩין יפנה מסגרות אלו למחייצה מיוחדת בדיסק – swap area.

מסגרות של מטען הדפים

- מכילות מידע שמקורו בקובץ.
- לדוגמה: אישור הזיכרון של הקוד.
- במידה וחסר זיכרון במערכת, הגሩין יפנה מסגרות אלו לקבצים בדיסק שמהם הן הגיעו.

בזיכרון הפיזי יש גם מסגרות של הגሩין (מכילות למשל ... PCB, runqueues ...).
אנו חנכו מתעלמים מהם כאן לצורך הפשטות.

טבלת המסגרות

- מערך עם כניסה לכל מסגרת בזיכרון הפיזי.
- כל כניסה במערך היא מסוג **struct page** ומכליה מספר שדות:
 - .1 – כמה מרחבי זיכרון מצביעים אל המסגרת?
- אם ערך המונה הוא 0, אפשר לפנות את המסגרת.

המנגנון COW שלמדו בשיעור
שעובר משתמש בשדה זה

– **mapping** .2

- מצביע ל-`inode` של הקובץ אם המסגרת שייכת למטען הדפים.
(`inode` נלמד בתרגול על מערכות קבצים).
- מצביע ריק (`NULL`) במקרה של מסגרת אונומית.

– **index** .3

- ההיסט מתחילה הקובץ (`offset`) עבור מסגרת של מטען הדפים.
- שדה ריק עבור מסגרת אונומית.

Struct page is defined in

https://elixir.bootlin.com/linux/v4.15/source/include/linux/mm_types.h.

There are actually two types of reference counts for a normal page, `refcount` and `mapcount`, but we will not go into these details.

טבלת המסגרות – שדות נוספים

- .4. דגלים המתארים את מצב המסגרת, כגון:
PG – מצין שתוכן המסגרת "מלוכלך", כלומר כתבו למסגרת בעבר.

איך ידועים אילו מסגרות מלוכלות?

- PG_referenced, PG_active – שומרים את רמת הפעילות (נראת בהמשך).
אלו למשה שני ביטים ולכך הערכיים האפשריים הם 0, 1, 2, או 3.

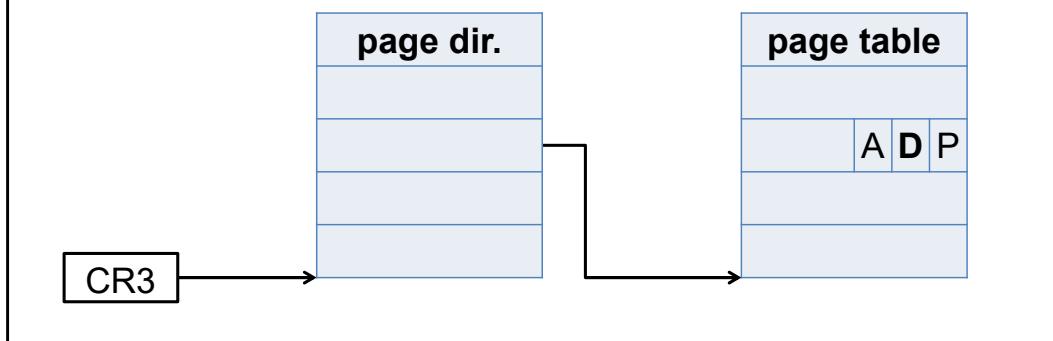
.5 – next hash, prev hash

- מצביעים למסגרת הבאה/הקודמת בשרשראת ההתגשויות של טבלת ערבות דפים
(נראת בהמשך).

התשובה בשקף הבא.

בֵּיט dirty בטבלת הדפים

- כasher המעבד כותב לדף מסוים הוא הולך בטבלת הדפים ואז מדליק את הביט dirty.
- אם התרגום של דף קיים ב-TLB אבל הביט dirty כבוי, המעבד ייר בטבלת הדפים כדי להדליק את הביט. הפעולה הזאת מתרוצעת ברקע ולא מעכבת את המעבד מלhmaשיך לפוקודה הבאה.



דוגמה: טבלת המסגרות

	refcount	flags	mapping	index
...				
#10	0	0	NULL	---
#11	3	AR	“/usr/lib/libc.so”	0
#12	2	A	NULL	---
#13	1	DR	“/home/assaf/file.txt”	5
#14	0	DA	“/home/dan/main.c”	0
...				

D = dirty
A = active
R = referenced

The mapping field is a pointer to the inode of the file. Here we write the filename instead for simplicity.

דוגמה: טבלת המסגרות

- בדוגמה מהש侃פ' הקודם:
- מסגרת 10 היא ריקה.
- מסגרת 11 מצביעה לבלוק הראשון בקובץ so./usr/lib/libc.so.
- שלושה תהליכיים שונים משתמשים ברגע במסגרת זהו.
- מסגרת 12 היא אוניבימית.
- שני תהליכיים משתפים ברגע את המסגרת זהו (למשל אבא ובן אחוי ()fork).
- מסגרת 13 מצביעה לבלוק השלישי בקובץ txt./home/assaf/file.txt.
- תהליך אחד בלבד משתמש ברגע במסגרת זהו.
- מסגרת 14 מצביעת לבלוק הראשון בקובץ c./home/dan/main.c.
- אף תהליך לא משתמש ברגע במסגרת זהו.

למה לינוקס לא מפנה מיד את המסגרת הזה?

תשובה: כי המסגרת הזה אולי תהיה שימושית בעתיד, למשל כאשר המשתמש ייצור תהליך חדש שקורא את הקובץ הזה.
באופן כללי, לינוקס לא ממהרת לפנות ذיכרון פיזי אלא עושה זאת רק אם חיברים (כי חסר זיכרון במערכת).

סיבוכיות גישה לטבלת המוגדרות

- נסמן ב- N את מספר המוגדרות הכלל.
- הכנסה ומחיקה של מוגדרת באינדקס ספציפי בטבלה – $(1)O$.
- מציאת מוגדרת ריקה – $(N)O$.
- כדי להקטין את הסיבוכיות, לנוקס משתמש במבנה נתונים נוספים.
- לדוגמה ה-buddy allocator – לא נלמד עלייו בקורס.
- חיפוש מוגדרת ספציפית של קובץ X בהיסט a – $(N)O$.
- כדי להקטין את הסיבוכיות, לנוקס משתמש בטבלה ערבות דפים (page hash table).

טבלת ערבול דפים

- טבלת ערבול דפים מספקת מיפוי מהזוג $(\text{mapping}, \text{index})$ לכתובת מסגרת (אם יש כזו) המכילה את הדף במיקום index של האובייקט mapping .
- כמו בכל טבלת ערבול, יכולות להיות "התנגשויות": זוגות שונים של $(\text{index}, \text{mapping})$ יכולים להתreffen לאותו אינדקס בטבלה המסגרות.
- לינוקס לחברת את כל הזוגות המתנגשים לרשימה מקוشرת כפולה מעגלית דרך השדות next_hash , prev_hash , next בטבלה המסגרות.
- כדי לחפש בטבלה ערבול דפים, לינוקס עוברת על כל המסגרות ברשימה ובודקת אם יש מסגרת עם $(\text{index}, \text{mapping})$ המבוקש.
- סיבוכיות החיפוש – $O(1)$ בממוצע.

דוגמה: טבלת ערבול דפים

טבלת ערבול דפים

	struct page *
#0	
...	
#50	
...	
#60	
...	
#70	
...	
#80	

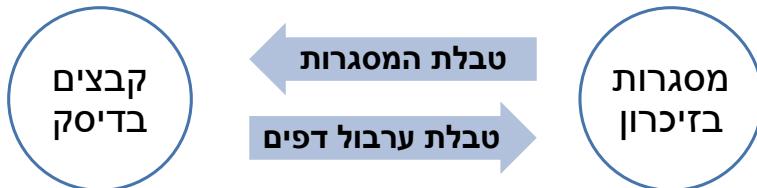
טבלת המסגרות

	next hash	prev hash	mapping	index
#0				
...				
#100			X	a
...				
#200			Y	b
...				
#300			Z	c
...				
#400			NULL	---



סיכון בינויים

- לינוקס שומרת מיפוי דו-כיווני:



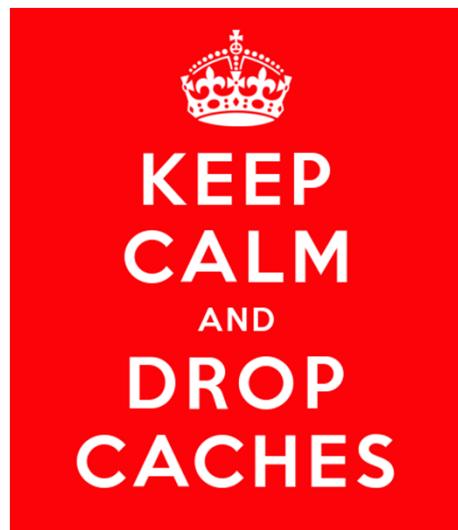
- שני מבני הנטוונים ממומשימים בתוכנה בלבד ללא תמיכת חומרה.

- איך מطمון הדפים משתמש במבנה הנטוונים הללו?

- כאשר לינוקס צריכה לקרוא דף מהדייסק, היא בודקת קודם בטבלת ערבות דפים אם המידע המבוקש כבר נמצא במטמון הדפים.
- כאשר לינוקס צריכה לפנות דף "מוליכך" לדיסק, היא בודקת בטבלת המסגרות את המיקום של הדף בדיסק.

כדי לפשט את התמונה, אונחנו מציגים את הדיסק כאילו הוא מחולק למגירות בגודל 8KB. בפועל, רק מאגר הדפודף (swap area) מחולק למגירות, ואילו שטח הדיסק של קבצים רגילים מחולק לבlokים בגודל 512B. לכן, כאשר מטמון הדפים מקשר מסגרת עם דף מסוים בקובץ, יש צורך בעוד רמת תרגום בין הדף לבlokים המרכיבים אותו – זה התרגום שנשמר ב-`swap` כפי שנראה בתרגול הבא.

הפרק



אופני גישה למטרון הדפים

בלינוקס יש שני אופני גישה לדיאק

מייפוי קובץ לזיכרון באמצעות
קריאה המערכת (`mmap`)

- קריאה המערכת יוצרת איזור זיכרון חדש ולא קוראת/כותבת מידע מהדיסק.
 - נוהל זיכרון עצל/דוחני.
- נסיוון גישה לזכרון יגרום לחריגת דף שתקרא את המידע מהדיסק למטען הדפים, ואז תעדקן את טבלת הדפים להציג יישורות למסגרות של מטען הדפים.

קריאה/כתיבה באמצעות קריאות המערכת (`read()`/`write()`)

- קריאה המערכת מביאה את המידע מהדיסק למטען הדפים, ואז מעטיקה את הנתונים הרלוונטיים אל או מהחיצים (`buffer`) של המשתמש.

תזכורת: שני אופני פעולה של mmap()

אנונימי

(anonymous)

```
mmap(..., fd=-1, ...);
```

- אזור הזיכרון מכיל מידע שאינו הקשור לשום קובץ, אלא לזיכרון הדינמי של התהילה.
- לדוגמה: אזור הזיכרון של המחסנית והערימה.

מוגבה קובץ

(file-backed)

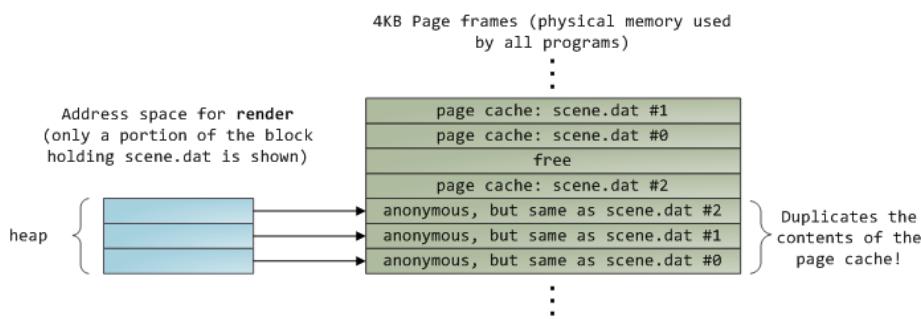
```
int fd=open("file", ...);
mmap(..., fd, ...);
```

- אזור הזיכרון מכיל מידע שמקורו בקובץ פתוח.
- לדוגמה: אזור הקוד, אשר מוגבה את הקובץ הבינארי של התוכנית.
- קריאה/כתיבה לאזור זיכרון מוגבה קובץ מתורגם לקריאה/כתיבה למיקום המתאים בתוך הקובץ.

הקוד – אזור זיכרון מוגבה קובץ,

גישה באמצעות write()/read()

```
int fd = open("scene.dat", O_RDONLY);
char* buffer = (char*)malloc(3*PAGE_SIZE);
read(fd, buffer, 3*PAGE_SIZE);
```



שימוש לב: הקוד המופיע בשקף יגרום ל-3 חריגות דף בהנחה שהחוצץ buffer הוקצה בצורה עצלה (למשל אם mmap קראה ל-(`mmap` CD) להקצותו אותו).

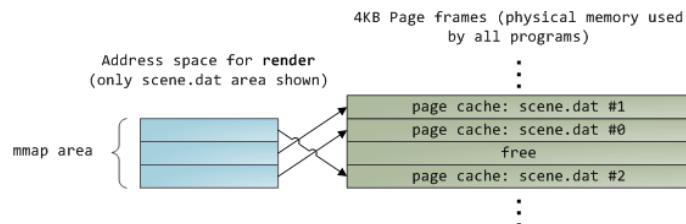
לכארה, נראה שאין יתרון ל-(`read` על-פני מיפוי לזיכרון באמצעות (`mmap`, כי שני אופני הגישה יגרמו ל-3 חריגות דף).

אבל אופן השימוש הנפוץ בקריאת המערכת (`read`) הוא הקציה חד-פעמית של החוצץ `buffer` ואז קריאה בלולאה של עוד ועוד מידע לתוך החוצץ.

במקרה הזה התקורה של חריגות דף להקצאת החוצץ `buffer` היא זניחה.

גישה באמצעות mmap()

```
int fd = open("scene.dat", O_RDONLY);
char *array = (char*)mmap(NULL, 3*PAGE_SIZE,
    PROT_READ, MAP_SHARED, fd, 0);
x = array[0]; // 1st page fault
x = array[0]; // no page fault
x = array[PAGE_SIZE-1]; // no page fault
y = array[PAGE_SIZE]; // 2nd page fault
z = array[2*PAGE_SIZE]; // 3rd page fault
```



יתרונות השימוש ב-`mmap` (לעומת `(read/write)`)

- **חסכון בזמן** – אין צורך להעתיק את המידע ממemoון הדפים לחיצים של התהילר.
- **חסכון בזיכרון** – אין חיצים למרחב המשתמש ולכן אין שכפול מידע שכבר קיים בזיכרון הדפים.
 - בנוסף, אם המידע משותף (נראה בשקפים הבאים), אז כל התהיליכים משתפים ביניהם את אותה מסגרת בזיכרון הפיזי.
- **ממשק פשוט וnoch לתוכנת** – ניתן לקובץ כפי שניגשים לזכרון.
 - קריית מערכת אחת במקום הרבה קרייאות מערכת `(read(), write())`.
- **קריאה דחיבנית** – העתקת המידע נעשית רק בעקבות ניסיון גישה שיוצר חריגת דף.
- שימוש לב: זה יכול להיות חיסכון של (`map()`, כפי ש谟וסבר בשקף הבא).

חסרונות השימוש ב-`mmap` (לעומת `(read/write)`)

- כדי למפות קובץ שלם יש למצאו איזור זיכרון פנוי ורציף למרחב הווירטואלי שהוא בגודל הקובץ.
- המגבלה הזאת לא באמת משמעותית במעבדי 64 ביט.

מה המגבלה על גודל הקובץ?

- קריאה המערכת `mmap` איטית יחסית ל-`read`.
- `read` עדיפה כאשר קוראים רק כמה בתים מהקובץ כי אז המחבר של קריאת המערכת `mmap + Chrigat Df` עלול להיות גדול יותר מזמן הגישה לדיסק.

תשובה: בערך 128TB, כי זה גודל מרחב הזיכרון הווירטואלי של המשתמש.

מיפוי משותף מול פרטי

מיפוי פרטי MAP_PRIVATE

- מטמון הדפים מחזיק עותק יחיד של המידע, אבל הוא מוגן באמצעות `copy-on-write`.
- בהתחלתה, כל המיפויים מצביעים לעותק זהה.
- כתיבות מצד תהליך כלשהו יגרמו לחריגת דף והעתקת המידע למסגרת חדשה.
- כתיבות לא יגעו חזרה לדיסק.

מיפוי משותף MAP_SHARED

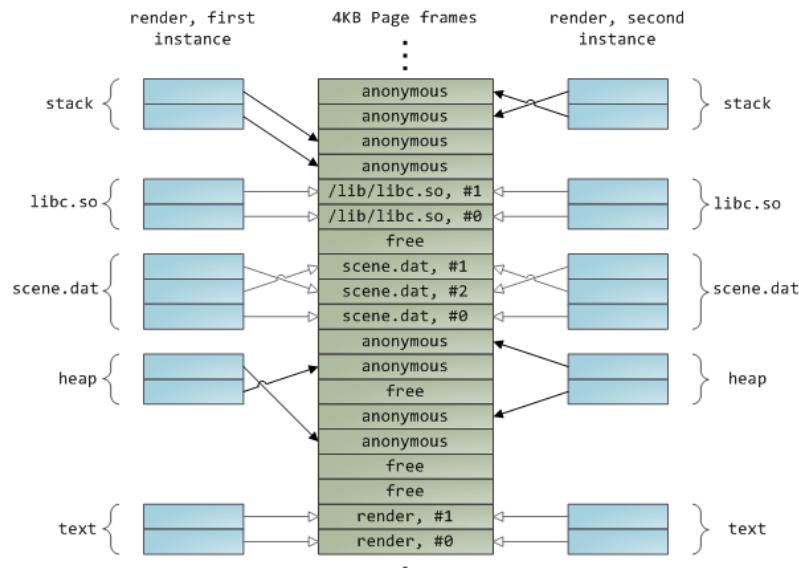
- מטמון הדפים מחזיק עותק יחיד של המידע.
- כל המיפויים המשותפים מצביעים לעותק זהה.
- כתיבות מצד תהליך כלשהו ייראו גם אצל תהליכיים אחרים הממפירים את אותו הקובץ.
- כתיבות לאיזור הזיכרון יחוללו בסופה של דבר לקובץ בדיסק.

שימוש לב: קריית המערכת (`mmap` ח'יבת לקלט אחד מהדgelים כדי לדעת האם המיפוי משותף או פרטי).

שיתוף זיכרון במתמון הדפים

- **תכניות שונות משתמשות לפעם בקבצים זהים.**
- **לדוגמה: כל התכניות הכתובות בשפת C משתמשות בספריה הדינמית `libc`.**
- **כדי לא לטעון עותקים זהים ומיתרים של הקובץ בזיכרון, מטמון הדפים שומר עותק יחיד וכל התהילכים מצביעים לעותק זה.**
- **כלומר, הكنيיות המתאימות בטבלת הדפים מצביעות לאותה המסגרת.**
- **למשל בشرطוט המופיע בשקף הבא, שני תהילכים המריצים את אותה התוכנית מצביעים לאותו עותק של הספריה `libc`, לאותו עותק של התוכנית (`render`), ולאותו עותק של קובץ אחר (`scene.dat`).**
- **כל המיפויים מוגדרים כפרטיים, כלומר הם מוגנים ע"י `COW`.**

שיתוף זיכרון במתמונן הדפים



from: <https://manybutfinite.com/post/page-cache-the-affair-between-memory-and-files/>

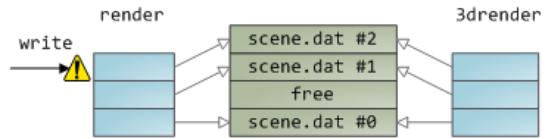
COW במתוחן הדפים

→ Page table entry marked read-only
 → Page table entry marked read/write

1. Two programs map scene.dat privately.
 Kernel deceives them and maps them both onto the page cache, but makes the PTEs read only.



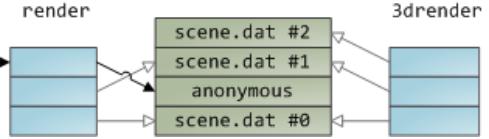
2. Render tries to write to a virtual page mapping scene.dat. Processor page faults.



3. Kernel allocates page frame, copies contents of scene.dat #2 into it, and maps the faulted page onto the new page frame.



4. Execution resumes. Neither program is aware anything happened.



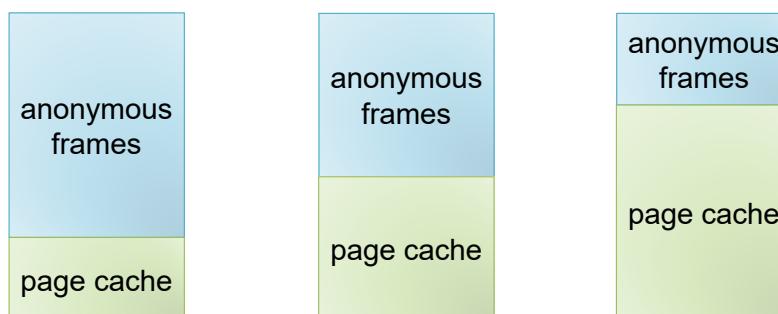
אלגוריתם שחרור מסגרות

Page Frame Reclamation Algorithm (PFRA)

בדומה לזמן התהילcis, האלגוריתם לשחרר מסגרות בלינוקס מבוסס על היוריסטיות. מכיוון שאין תיאוריה מפותחת בתחום של אלגוריתמי שחרור דפים, המפתחים של לינוקס ממשיכים לבחון רעיונות חדשים בשיטה של ניסוי וטעייה.

מה גודל מטמון דפים?

- מצד אחד, תהליכי שನיגשים הרבה לדיסק יעדיפו מטמון דפים גדול כדי לחסוך גישות יקרות לדיסק.
- מצד שני, תהליכי חישוביים יעדיפו מטמון דפים קטן כדי שלא יבוא על חשבון זיכרון אונוניי לתהליכי.
- מעט זיכרון לתהליכי ← יותר גישות לדיסק.



במצב הקיצון של מטמון דפים קטן מדי עלולה לקרות תופעת **thrashing** – מצב בו המערכת מבלה את רוב הזמן במענה לחריגות דף והעתקת מידע בין הדיסק לזכרון במקום בביצוע התהליכים.

מה גודל מטמון הדפים?

- בلينוקס אין חסם על גודל מטמון הדפים.
- כל בקשה להוסיף מסגרת למטמון הדפים בעניית בחיוב.
- כאשר יש הרבה זיכרון פנוי ניתן לנצל אותו כדי להגדיל את מטמון הדפים וכך לשפר את הביצועים שלו.
- גם בקשות של תהליכייה להקצת מסגרת אונונימית בענותות תמיד בחיוב.
- אבל מה קורה כאשר אין יותר מסגרות פנויות?

אלגוריתם שחרור מסגרות

- כasher לא נותרו עוד מסגרות פנויות, הגרעין קורא לאלגוריתם שחרור מסגרות (PFRA = Page Frame Reclamation Algorithm).
- מסגרות של מטמון הדפים יפנו חזרה לקבצים המתאים בדיסק רק אם הן מסומנות dirty, כלומר אם נכתב אליהן מידע חדש.
- אם המסגרת נקייה אין צורך לכתוב אותה שוב לדיסק כי הקובץ המקורי ממנו הגיעו המסגרת כבר שומר אותה.
- מסגרות אונונימיות יפנו לאיזור מיוחד בדיסק הנקרא מאגר דף-דף. מסגרת אונונימית תמיד תפנה לדיסק, בין אם היא מalloc'ת או נקייה, בגלל שהיא נמצאת בשימוש של תהליך פעיל.
- שימו לב: PFRA מפנה רק מסגרות של תהיליכי משתמש, ולעתים לא מפנה מסגרות בשימוש הגרעין.

למעשה האלגוריתם מפנה גם סוג שלישי של מסגרות –ائلו השיקות למיטמוני בזיכרון כמו slab cache, inode cache, dentry cache . פינוי מסגרות מהסוג השלישי לא דורש שום כתיבה לדיסק. אנחנו לא נלמד על המיטמוניים הללו בקורס זהה.



מאגרי דפודף בלינוקו

- מאגר דפודף (swap area) הוא אזור מיוחד בדיסק אליו מפונים דפים מהזיכרון.

- לינוקס מאפשרת להגדיר מספר מאגרי דפודף, ובנוסף ניתן להפעיל ולכבות מאגרי דפודף באופן דינמי תוך כדי פעולה המערכת.

מה היתרונות של מספר מאגרי דפודף שונים?

- כל מאגר דפודף הוא שטח דיסק המחולק למגריות (slots).

- כל מגירה היא בדיק בגודל דף / מסגרת (4KB).

- המגריה הראשונה מכילה מידע ניהולי על המאגר: גודל, גרסה, וכו'.

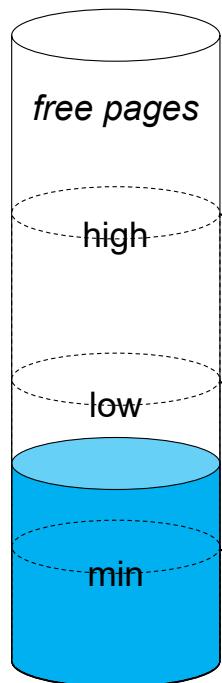
- אלגוריתם הדפודף משתמש בנקודת מגירות ברצף לדפים מפונים.

למה עדיף ברצף?

תשובה 1#: שיפור ביצועים של swapping ע"י כתיבה במקביל למספר מאגרי דפודף.

תשובה 2#: כדי לשפר את זמן הגישה לדיסק (גישה סדרתית היא מהירה יותר – מפחית סיבובים של הזרוע המגנטית, משפר את seek latency).

מתי מפנים זיכרון?



- כאשר **high** $>$ **pages_high** PFRA מפסיק לפעול.
- כאשר **low** $<$ **pages_low** PFRA מתחילה לפעול באופן אסינכריוני (ברקע).
- כאשר **min** $<$ **pages_min** PFRA מתחילה לפעול באופן סינכריוני (בhzית).

שני אופנים לפינוי זיכרון

פינוי סינכרוני

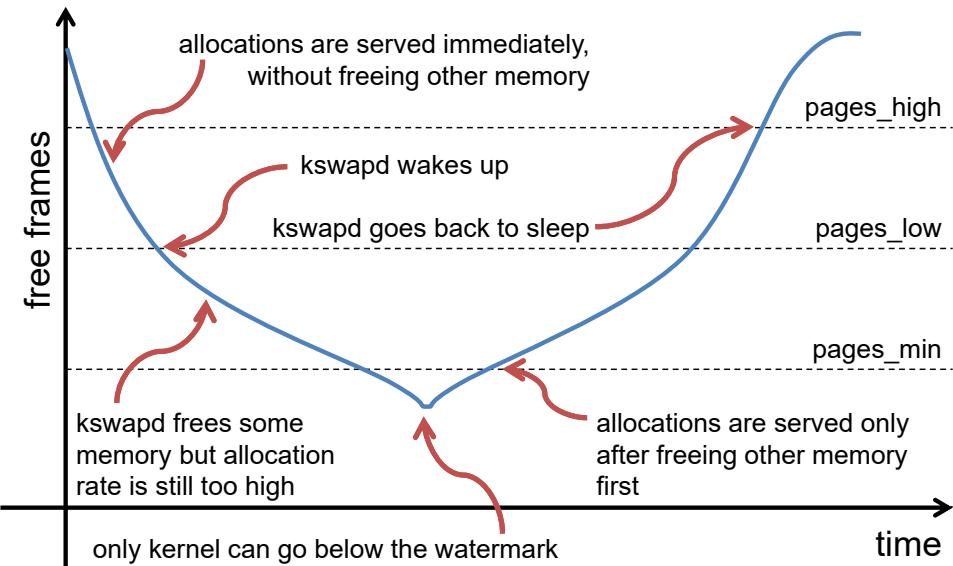
- אבל kswapd לא בהכרח מפנה זיכרון מספיק מהר...
- אם במהלך הקצאת זיכרון הגרעין מגלה שמספר המסגרות הפנויות קטן מהסף הקרייטי אז PFRA נקרא ישירות.
- במקרה זה התהילך ש策יר את הזיכרון נאלץ להמתין.

פינוי אסינכרוני

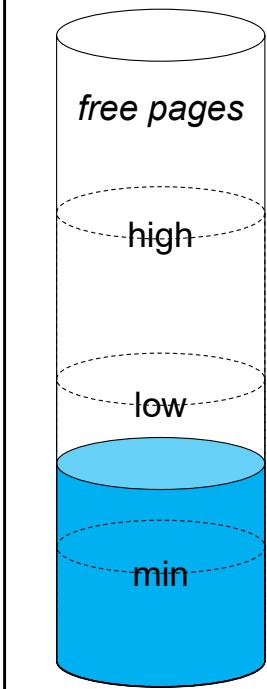
- אם במהלך הקצאת זיכרון הגרעין מגלה שמספר המסגרות הפנויות נמוך מכך – הוא מעיר חוט גרעין מיוחד – kswapd – שתפקידו להרייז את PFRA.
- kswapd רץ ברקע ומפנה זיכרון במקביל לתוכניות אחרות אשר מבקשות זיכרון.
- יעיל במקרים רבים מעבדים.

Do not confuse kswapd, the kernel thread that is responsible for page reclaiming, with the swapper process (PID==0), which is called swapper for historical reasons (swapper no longer performs memory management).

תרחיש לדוגמה



ספ מינימום, ספ תחתון, וספ עליון



- מדוע מוגדר ספ מינימום קרייטי (high)?
- PFRA דורש זיכרון פנוי כדי לרווח, למשל כדי לתמוך מבני נתונים של הגרעין. אם מספר המסהגרות הפנויות ירד מתחת לסקף הקרייטי, הגרעין עלול לקריסות.
- מדוע מוגדר ספ תחתון (low)?
 - כדי להקטין את הזמן של הקצאה חדשה.
- הגרעין פינה מראש בזיכרון אסינכרונית כדי שהיה זיכרון מעבר לסקף הקרייטי וכן אין צורך לפנות מסגרות בזיכרון סינכרונית.
- מדוע מוגדר ספ עליון (high)?
 - כדי לא לפנות יותר מדי דפים ולפגוע בביצועים.
 - כדי לא לבצע עבודה מיותרת.

עקרונות כלליים של PFRA

- העקרונות המרכזיים המשותפים לכל המימושים של PFRA:
 - .1. בחינת כל המסגרות של כל התהיליכים ועדיפות לפינוי מסגרות של מטען הדפים שאף תחילר לא מצביע אליון /או מסגרות נקיות.
 - .2. מעקב דינמי אחר רמת הפעולות של כל המסגרות ועדיפות לפינוי המסגרות ה"קרות" ביותר.
 - תהליכי משתמש שיוצאים להמתנה ארוכה יאבדו בהדרגה את כל המסגרות שהם חזיקו ולא יפריעו לשאר התהיליכים.
 - .3. פינוי מסגרת משותפת מכל מרחב הזיכרון המצביעים עליה בבת-אחת.

סיווג מסגרות של PFRA

מסגרות של
מטען הדפים

מלוכלות
mozbe'ot

נקיות
mozbe'ot

מלוכלות
la mozbe'ot

נקיות
la mozbe'ot

מסגרות
anonimiyot

mozbe'ot

- מסגרות לא Mozbe'ot – מסגרות שאף טבלת דפים (כלומר אף תהילר) לא מצביעה עליהם.
- מדוע אין מסגרות anonimiyot לא Mozbe'ot?
- כי מסגרות anonimiyot נמחקקות מיד עם סיום התהילר המצביע עליהם, ככלומר מיד כאשר הן לא Mozbe'ot.

אילו מסגרות עדיף לשחרר?

1. מסגרות לא מוצבעות – כי פינוי מסגרות מוצבעות דורש עדכון טבלאות הדפים של תהליכי המשמש המוצבעים עלייהן.
2. מסגרות נקיות – כי פינוי מסגרות מлокלכות דורש כתיבה חזקה לדיסק.
 - האם תמיד עדיף לשחרר מסגרות לא מוצבעות על-פני מסגרות אונונימיות?
 - לא. לדוגמה: תכנית אחת שעורכת קוד ותכנית שנייה שמהדרת (מקומפלט) אותו.
 - אחרי שהתכנית הראשונה מסתייםמת ולפניהם שהתכנית השנייה מתחילה, המסגרת של הקובץ לא מוצבעת, אבל כדאי לשמור אותה בזיכרון למשך זאת.

כדי לפתור את הבעיה המוצגת בסוף השקף, ניתן מציגה את הפרמטר `swappiness`, הנשלט ע"י המשתמש ומאפשר לו לכוון את היחס בין פינוי מסגרות של מטען הדפים ומסגרות אונונימיות.

<https://lwn.net/Articles/690079/>

אלגוריתם פינוי מסגרות אופטימלי

- האלגוריתם האופטימלי (== מביא למספר ההחטאות הנמוך ביותר)
יפנה את המסגרות שניגש אליהן בעתיד הרחוק ביותר.
.Bélády's optimal page replacement policy •
- מכיוון שלא ניתן לדעת את העתיד, PFRA מסתכל על העבר ומנסה
לחזות ממנו את העתיד:
 - דפים שניגשו אליהם לאחרונה – הצפי הוא שייגשו אליהם שוב
בעתיד הקרוב.
 - דפים שניגשו אליהם לפני זמן רב – הצפי הוא שייגשו אליהם שוב רק
בעתיד הרחוק.
- בדומה לאלגוריתם LRU:Least Recently Used

מעקב אחרי גישה לדפים

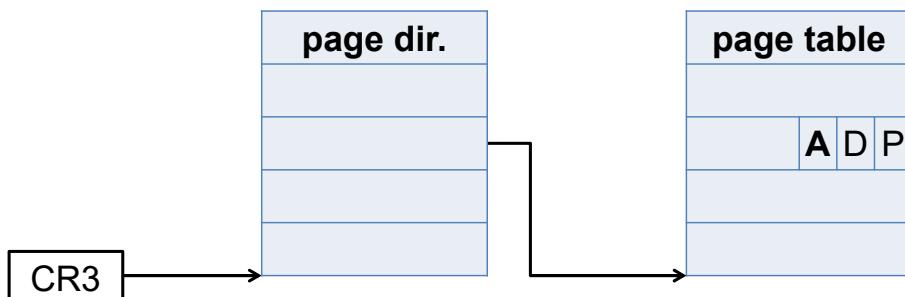
- כדי למש את האלגוריתם שתואר בשקף הקודם, מערכת הפעלה צריכה לדעת על כל גישה לכל דף בזיכרון.
- אפשרייה 1#:** המעבד ייצור פסיקה בכל גישה לזכרון כדי להעביר את השליטה למערכת הפעלה.
 - זה פתרון לא סביר כי המעבד ניגש לזכרון בכל פקודה לפחות פעם אחת (כדי לקרוא את הפקודה), ולכן התקורה תהיה בלתי נסבלת.
- אפשרייה 2#:** המעבד יתחזק בעצמו מונה לכל מסגרת המכיל את ה"גיל" שלה.
 - זה פתרון יקר בחומרה ולכן מעבדי אינטל/AMD לא מספקים תמיינה כזו.
- אפשרייה 3#:** מערכת הפעלה תשתמש במנגנון חומרה אחר כדי להעיר/לקרב את רמת הפעולות של המסגרות בזיכרון: ביט accessed של הכניסות בטבלת הדפים.

ביט accessed בטבלת הדפים

- כאשר המעבד ניגש לדף מסוים לראשונה, הוא מתרגם את הכתובת הווירטואלית שלו לכתובת פיזית ע"י הליכה בטבלת הדפים ואז מדליק את הביט accessed.

ומה אם התרגום של הדף נמצא ב-TLB?

- גישה נוספת לדף לא ישנו את מצב הדגל accessed.



תשובה: אם התרגום של הדף כבר נמצא ב-TLB אז סימן שהמעבד כבר ניגש לדף זהה בעבר ומצא את התרגום שלו בטבלת הדפים.
במקרה זה ברור שהדגל accessed כבר הודלק בעבר.

רמת פעילות של מסגרת

- שני הביטים active, referenced מגדירים לכל מסגרת בטבלה המסגירות את רמת הפעילות שלה: ערךשלם בין 0 ל-3.
- לדוגמה:

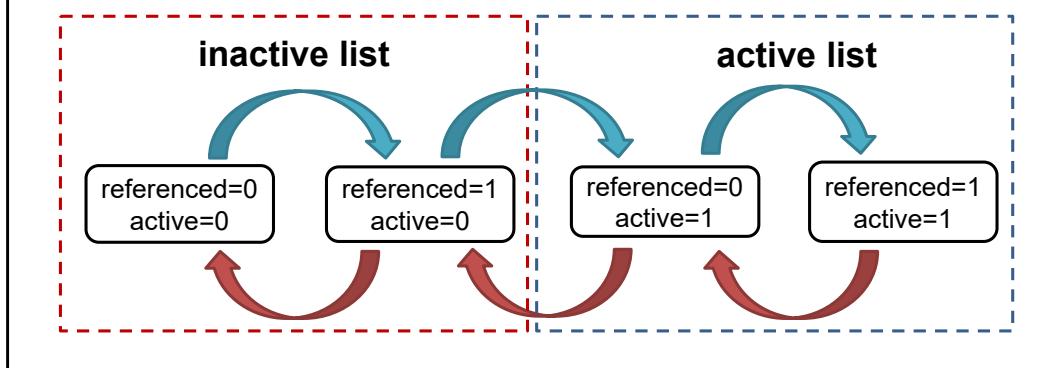
	refcount	flags	mapping	index
...				
#10		A,R=0,0 → activity = 0		
#11		A,R=0,1 → activity = 1		
#12		A,R=1,0 → activity = 2		
#13		A,R=1,1 → activity = 3		
...				

עדכן רמת הפעולות

- בכל פרק זמן מסוים, מערכת הפעלה סורקת את טבלאות הדפים ובודקת את מצב הדגל `accessed` בכל הכנסיות.
- בכל כניסה שבה הדגל `accessed` דלוק, מערכת הפעלה מגדילה את רמת הפעולות של המסגרת (`activity++`) ומנקה את הדגל `accessed`.
- בנוסף, בכל פרק זמן מסוים, או כאשר מספר המסגרות הפנויות נמוך, מערכת הפעלה מקטינה את מוני הפעולות של כל המסגרות (`activity--`).
- הרצינול: כדי לא להתחשב בהיסטוריה רחוקה מדי. אם מסגרת הייתה פעולה בעבר, זה לא אומר שהיא עדין פעולה בהווה.

שחרור מסגרות שלא היו בשימוש לאחרונה

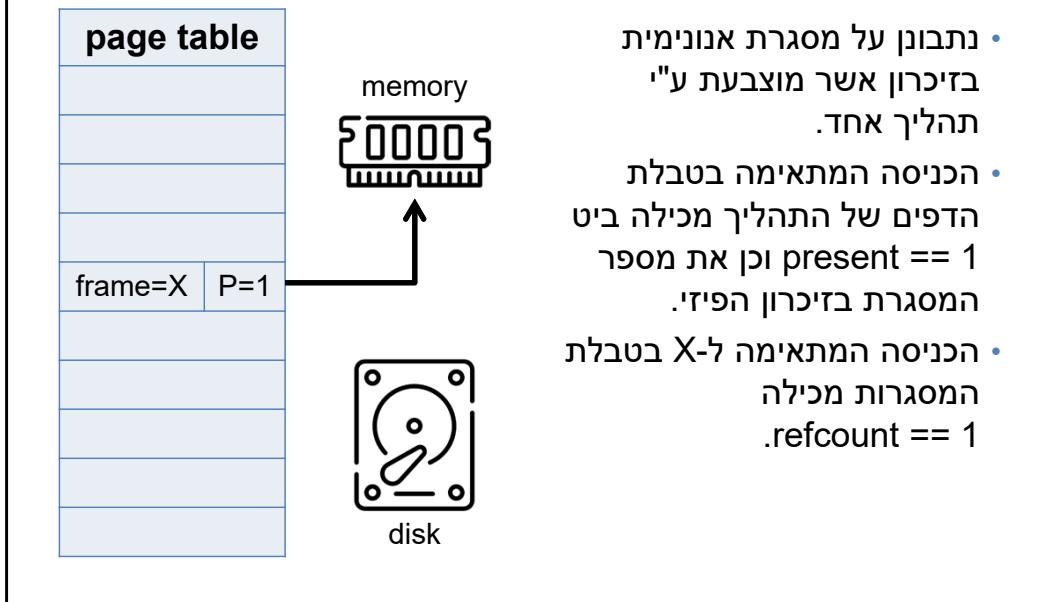
- כל מסגרת בזיכרון שיכת לאחת מבין שתי רשימות הקשורות (active, inactive) בהתאם לרמת הפעילות שלה.
- מסגרות יכולות לעבור במצב דינמי בין הרשימות.
- PFRA סורק את המסגרות ברשימה inactive ומפנה אותן קודם.



פינוי מסגרות משותפות

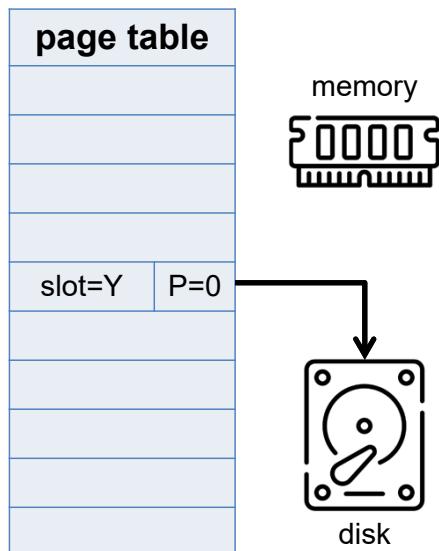
- מסגרות משותפות הן מסגרות המוצבעות מספר מרחבי זיכרון בו-זמןית.
 - דוגמה #1: מסגרת במתมอง הדפים אשר מוצבעות מספר תהליכיים שסימפים את אותו הקובץ לדיכרן.
 - דוגמה #2: מסגרת אונומית המשותפת בגלל מנגן WO (לפני שבוצע נסיוון כתיבה למסגרת).
- כאשר PFRA נדרש לפנות מסגרת משותפת, הוא מנטק אותה מכל מרחבי הזיכרון המוצבעים עליה בבהת-אחת.
 - כדי למש את האלגוריתם, ינוקס שומרת מיפי הפוך (מבנה נתונים שלא נלמד עליו בקורס) מכל מסגרת פיזית אל מרחבי הזיכרון המוצבעים עליה.
 - PFRA עובר על כל מרחבי הזיכרון הללו ומעדכן את הكنيסות המתאימות בטבלאות הדפים.

דוגמה מסכמת 1



Icon made by Freepik from www.flaticon.com

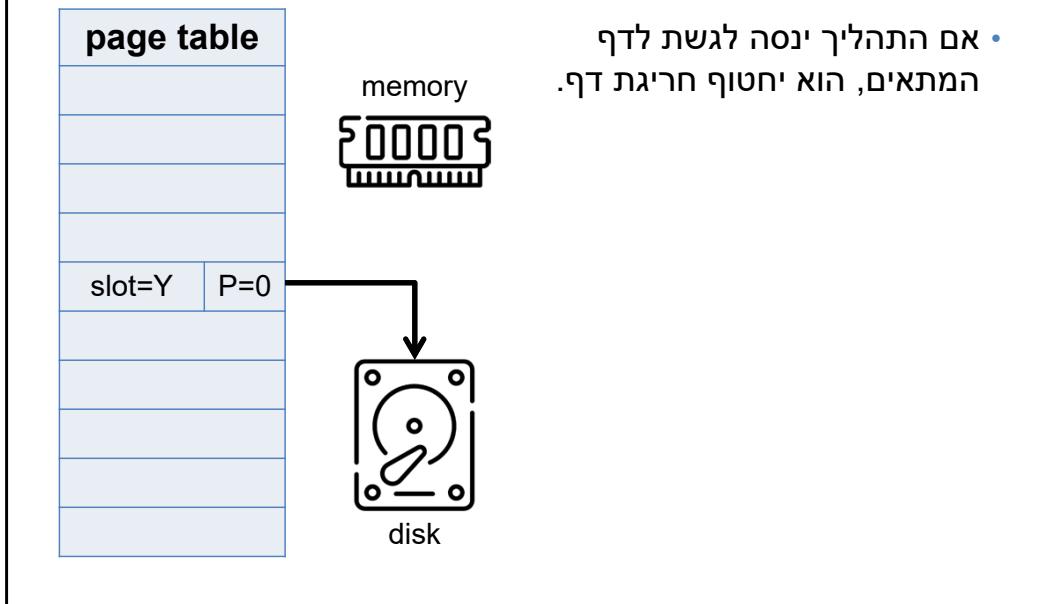
דוגמה מסכמת 2



- כעת נניח ש- PFRA מחליט לפנות את המסגרת לדיסק.
- הגרעין מחפש מגירה פנוייה במאגר דפודף כלשהו ומעתיק אותה את המסגרת.
- הכניסה המתאימה בטבלת הדפים של התהלייר מכילה ביט present == 0 וכן את מקום המסגרת בדיסק.
- הכניסה המתאימה ל-X בטבלת המסגרות מכילה .refcount == 0.

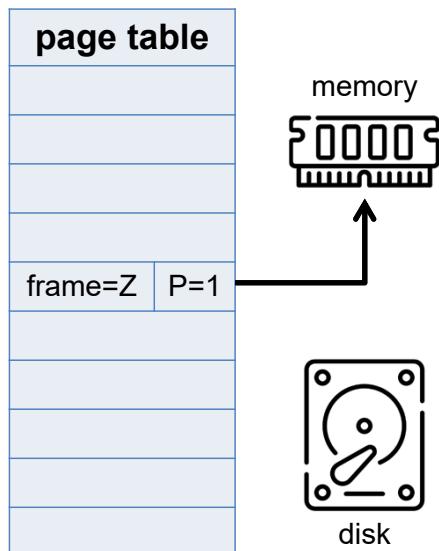
Icon made by Freepik from www.flaticon.com

דוגמה מסכמת 3



Icon made by Freepik from www.flaticon.com

דוגמה מסכמת 4



- הטיפול יעבור לארען, שיקרא את המסגרת חזורה לזיכרון – לא בהכרח למיקום הקודם שלו.
- הכניסה המתאימה בטבלת הדפים של התהלייר תציבע אל מיקום המסגרת בזיכרון.
- הכניסה המתאימה ל-Z בטבלת המסגרות מכילה `refcount == 1`.

Icon made by Freepik from www.flaticon.com

תרגול 13

ממשק מערכת הקבצים בLinux
Very Simple File System (VSFS)
File Allocation Table (FAT)

TL;DR

- user applications
 - system calls (open, read, write, ...)
- file system interface
 - inodes, dirent, ...
- concrete file systems
 - ext2, ext3, FAT32, NTFS, btrfs, ...
- page cache
- device drivers

- מצד אחד, בلينوكס .“everything is a file”
regular files, directories, links, •
sockets, pipes, fifos, ...
- מצד שני, קיימים מגוון סוגים
קבצים, אמצעי אחסון פיזיים,
וממערכות קבצים המציגות
משמעותיים שונים.
- لينوكס מוסיף שכבת
אבסטרקציה כדי להציג
למשתמש ממשך אחיד ונוח
למערכת הקבצים.

ממשק מערכת הקבצים בלינוקס

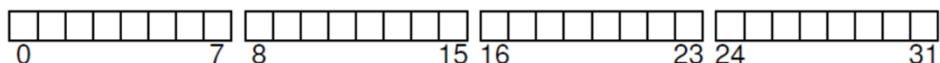
תזכורת: מהי מערכת הפעלה?

- מערכת תוכנה אשר אחראית על ניהול החומרה עבור המשתמשים.
- מערכת ההפעלה מספקת אבסטרקציות לשלוות רכיבי החומרה המרכזיים:

רכיב חומרה	\Leftrightarrow	אבסטרקציה עיקרית
معالד	\Leftrightarrow	תהליך
זיכרון פיזי	\Leftrightarrow	מרחב זיכרון וירטואלי
Disk	\Leftrightarrow	מערכת הקבצים

מהו דיסק?

- כונן דיסק קשיח (HDD = hard disk drive) הוא התקן אחסון עמיד.
- כלומר המידע נותר על הדיסק גם אם אין מתח, ובפרט גם לאחר כיבוי המחשב.
- מבחינת מערכת הפעלה, הדיסק הוא מערך של **סקטורים**.
- סקטור == 512 בתים רציפים המתחילה בכתובת מיושרת.



- למרות שהדיסק פועל ביחידות של סקטורים (512 בתים), מערכת ההפעלה מנהלת קבצים ביחידות גדולות יותר – **בלוקים** בגודל 4KB.

מדוע?

תשובה: 4KB זהו גודל הדף/מסגרת במערכת הזיכרון הווירטואלי.
מכיוון שמיידע עובר בין הדיסק לזיכרון (ולהיפר) באופן קבוע, נוח יותר למערכת הפעלה
לנהל את הזיכרון והדיסק באותה גרגנולריות.

מהו סקטור?

- הדיםק מבטיח ל מערכת הפעלה כי פועלות כתיבה של סקטור בודד היא אוטומית – הסקטור נכתב לדיסק בשלמותו, או לא נכתב בכלל.
- אין אוטומיות ביחידות גדולות יותר מסקטור בודד.
- כתבה של בלוק בגודל 4KB יכולה להיקטע באמצעות (למשל בגלל נפילה מתח) ואז המשמש יראה תוכנות כתיבה חילונית.
- בגלל מבנה הדיסק, גישה סדרתית לסקטורים סמוכים מהירה יותר (בערך פי 100) מגישה אקראית לסקטורים המפוזרים בדיסק.
- שימוש לבן: המונחים סקטורים ובלוקים מתבלבלים לעיתים...

מהו קובץ?

- מערך של בתים (ללא מבנה מיוחד).
- אוסף הפעולות על קבצים בלינוקס ניתן ע"י קריאות המערכת:
 - creat() – ייצירת קובץ חדש.
 - open() – פתיחת קובץ קיים (או ייצירת קובץ חדש).
 - read() – קריאה מתוך קובץ פתוח.
 - write() – כתיבה לקובץ פתוח.
 - close() – סגירת קובץ פתוח.
- (unlink – מחיקת קישור לקובץ (ואולי גם את הקובץ עצמו אם זה הקישור האחרון).
- עוד عشرות רבות של קריאות מערכת...

על קריאות המערכת (open()/close()/read()/write()) כבר דיברנו הרבה במהלך הקורס, ולכן לא נדבר עליו שוב כאן.

תכונות של קבצים

- מערכת הקבצים שומרת לכל קובץ גם תכונות נוספות (metadata) במבנה הנקרא `inode`. תכונות לדוגמה:
 1. `inode number` – מזהה ייחודי לקובץ.
 2. גודל הקובץ.
 3. הרשות גישה – למי מותר לקרוא/לכתוב/להריץ את הקובץ.
 4. חותמות זמן – הזמן האחרון בו קראו/כתבו מהקובץ.
 5. מקום בדיסק – הסקטורים בדיסק המרכיבים את הקובץ.
- במשר נראה איך מערכת קבצים שונות שומרת את המידע הזה.

שימוש לב: ה-`inode` אינו שומר את שם הקובץ כי לאוינו קובץ יכולם להיות מספר שמות שונים באמצעות קישורים (links).

שם הקובץ (ליתר דיוק, שם הקישור) נשמר בתיקיה המכילה אותו.

קריאה המערכת (stat())

```
user@ubuntu:~$ touch file
user@ubuntu:~$ stat file
  File: file
  Size: 0      Blocks: 0      IO Block: 4096   regular empty file
Device: 801h/2049d  Inode: 18483362      Links: 1
Access: (0644/-rw-r--r--)
Modify: 2020-01-12 10:44:38.006213596 +0200
Change: 2020-01-12 10:44:38.006213596 +0200
user@ubuntu:~$ echo -n "a" >> file
user@ubuntu:~$ stat file
  File: file
  Size: 1      Blocks: 8      IO Block: 4096   regular file
Device: 801h/2049d  Inode: 18483362      Links: 1
Access: (0644/-rw-r--r--)
Modify: 2020-01-12 10:44:38.006213596 +0200
Change: 2020-01-12 10:45:01.588753840 +0200
```

למה מספר הבלוקים הוא 8?

תשובה: מספר הבלוקים המופיע בשקף הוא בעצם מספר סקטוריים על הדיסק.
8 סקטוריים בני 512B הם בדיק בлок אחד בגודל 4KB, שהוא הגודל המינימלי לקובץ במקרה שלנו.

מהי תיקיה?

name	inode number
.	33
..	15
foo	5
bar	806
	-1
	-1
	-1

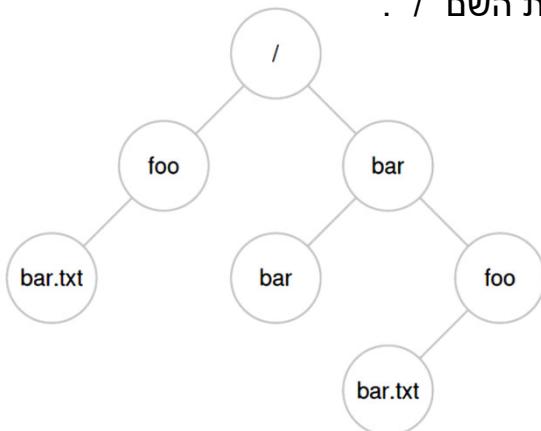
- סוג מיוחד של קובץ המוצג ע"י מערך של רשומות.
- כל רשומה היא מיפוי: name → inode number
- כל תיקיה מכילה תמיד שתי רשומות מיוחדות:
- ".." (נקודה) – מצביע לתיקיה הנוכחית.
- "..."(שתי נקודות) – מצביע לתיקיה האב.

אם התיקיה הנוכחית היא תיקיית השורש אז ".." מצביע לתיקיות השורש.

היררכית מערכות הקבצים

- כאמור, תיקיות בלינוקס מחייבת לקבצים ו/או תיקיות נוספים.
- לינוקס מארגנת את כל הקבצים והתיקיות לעץ ייחיד.
- שורש העץ הוא התיקיה בעלת השם "/" .

- ניתן להתייחס לקבצים בעץ לפי הנתיב האבסולוטי (absolute pathname) או לפי הנתיב היחסי (relative pathname) לתקינה הנוכחית.



פעולות על תיקיות

- ניתן ליצור תיקיה חדשה באמצעות קריית המערכת (`mkdir()`).
- ניתן להסיר תיקיה ריקה באמצעות קריית המערכת (`rmdir()`).
- לא ניתן לקרוא/לכתוב לקובץ תיקיה באמצעות קריאות המערכת (`(read|write)(...)` מכיוון שפרטיו המימוש של תיקיות מוסתרים מהמשתמש.
- מערכות קבצים שונות ממושכותתיקיות באופן שונה, כפי שנראה בהמשך.
- ניתן לקרוא קובץ תיקיה (כלומר לקרוא את רשימת הקבצים השייכים לתיקיה) רק באמצעות קריית המערכת (`getdents()`).
- ניתן לכתוב לקובץ תיקיה רק באמצעות יצרה של קובץ חדש או קישור חדש בתוכה.

בלינוקס יש שני סוגי קישורים (links)

soft / symbolic link

`In -s src dst`

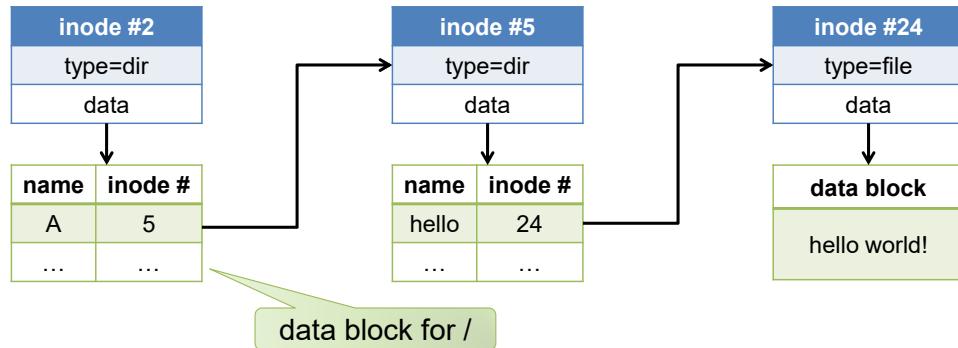
- קישור סימבולי הוא קובץ חדש עם שמו נפרד מזה של הקובץ המקורי.
- כתיבה דרך הקישור כתבתת לקובץ אליו הוא מצביע.
- מהיקת הקישור (באמצעות הפקודה `rm`) לא תמחק את הקובץ המצביע.
- אפשר ליצור קישורים סימבוליים גם לקובץ שלא קיים.

hard link

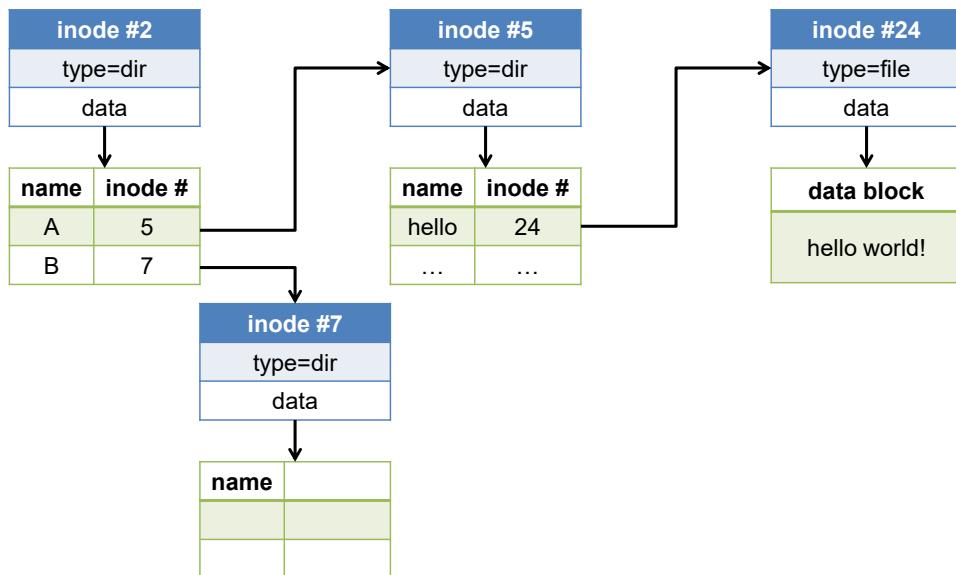
`In src dst`

- קישור קשיח הוא שם נרדף לקובץ המקורי כי הוא מצביע ישירות ל-`inode` של הקובץ המקורי.
- כתיבה דרך הקישור כתבתת לקובץ אליו הוא מצביע.
- מהיקת הקישור תקטין את מונה הקישורים של הקובץ (כפי שנשמר ב-`inode`).
- הקובץ ימחק מהדיסק רק כאשר כל ה-links hard אליו ימחקו.

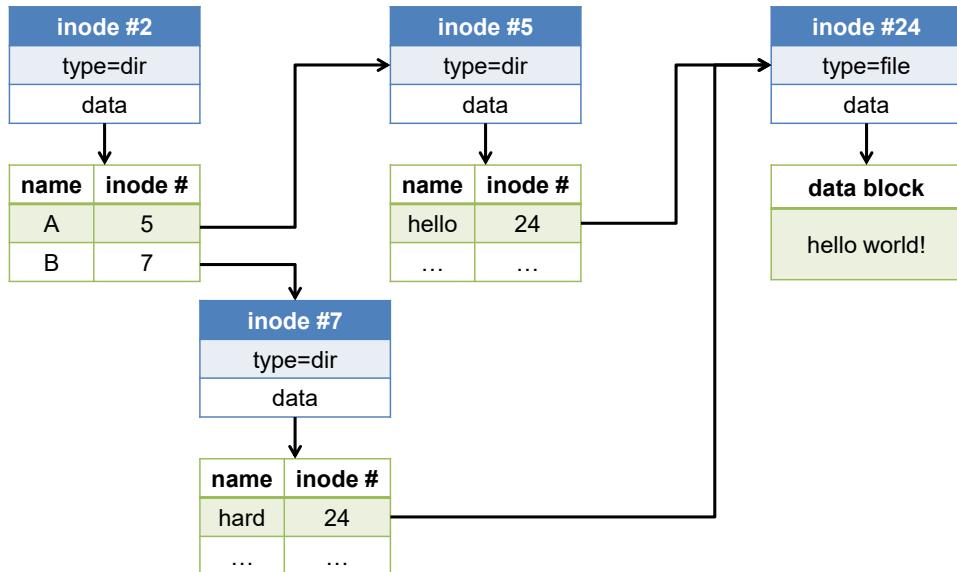
דוגמה: קישורים



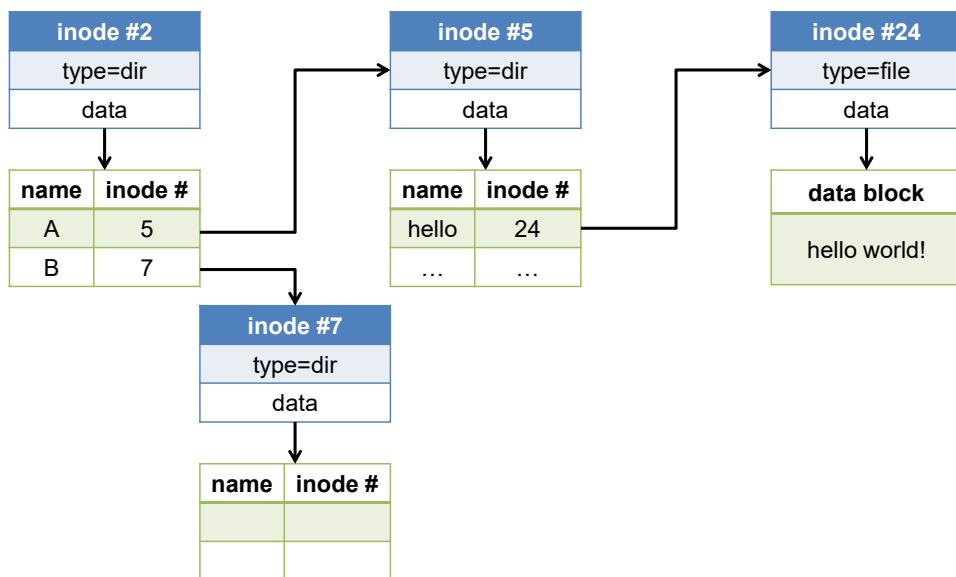
>> mkdir /B



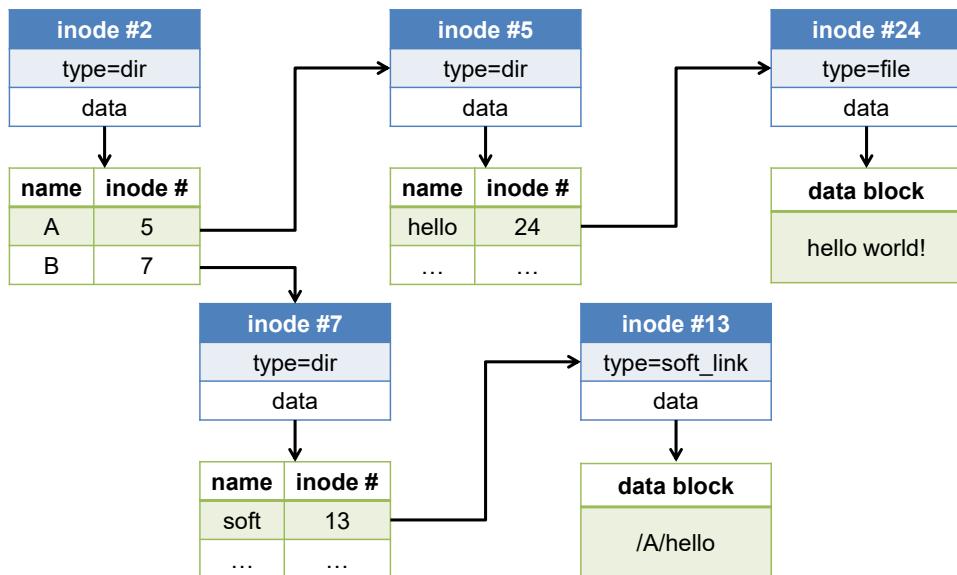
```
>> ln /A/hello /B/hard
```



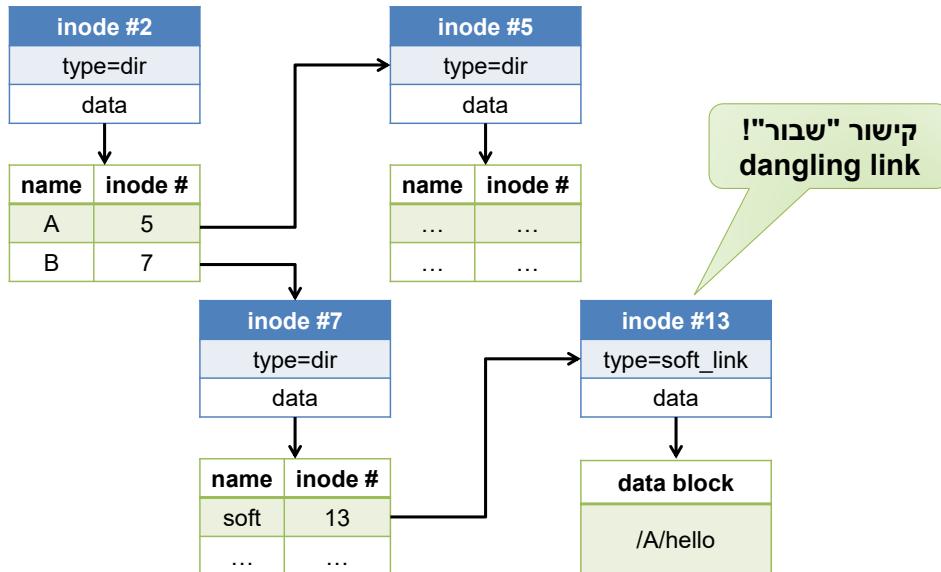
>> rm /B/hard



```
>> ln -s /A/Hello /B/soft
```



```
>> rm /A/hello
```



השואיה בין סוגי קישורים בלבד

קישורים רכימ

- ✓ יכולם לקשר בין שתי מערכות קבצים שונות (הקיימות על שני דיסקים נפרדים למשל).
- ✓ יכולם להצביע על תיקיות.
- ✗ יכולם להיות "שבורים".
- ✗ עשויים ליצור מעגלים במערכת הקבצים.

קישורים קשיים

- ✓ לא יכולים להיות "שבורים".
- ✗ לא יכולים להצביע על תיקיות – ראו שkopית הבהה.
- ✗ לא יכולים לקשר בין שתי מערכות קבצים שונות (הקיימות על שני דיסקים נפרדים, למשל).

מדוע?

Answer: inode numbers are not unique across file systems.

הצבעה לתקינות – מה הבעיה?

איןסוף מסלולים לאותו קובץ

- פעולה מסויימת עלולות להוביל לרקורסיה אינסופית, למשל הדפסת כל הקבצים תחת התקינה הנוכחי:

`/A/loop`

`/A/loop/loop`

`/A/loop/loop/loop`

...
הבעיה היא עדין קיימת בקישורים רכימ.

לינוקס מונעת אותה באמצעות ציהוי מעגליים בגרף היררכיות הקבצים.

בלבול קשיי המשפחה

- אם קישורים יצביעו לתקינות:
`>> cd /A`
`>> ls /A/loop`
- אז לא ניתן להגדיר תקינות אב (parent directory) אחת יחידה לכל קובץ במערכת.
- בדוגמה מעלה: `A/` היא תקינה האב של `loop`, אךvr גם `loop/A/`.

Further reading: <https://askubuntu.com/questions/210741/why-are-hard-links-not-allowed-for-directories>

VERY SIMPLE FILE SYSTEM (VSFS)

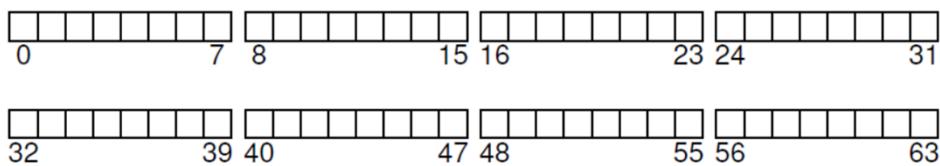
הסטודנטים לומדים גם בהרצאה על מערכת הקבצים הzo, אבל בצורה שטחית יחסית.
החומרים בפרק זה מבוסס על פרק 40 בספר OSSTEP.

Very Simple File System (VSFS)

- CUT נבנה בצורה הדרגתית מערכת קבצים פשוטה בשם VSFS.
- מערכת הקבצים זהו היא גרסה מופשטת של מערכת הקבצים המקורי של UNIX ולכן היא מציגה באופן פשוט את המושגים הבסיסיים של מערכות קבצים בلينוקס.
- מערכת הקבצים ממומשת בתוכנה בלבד, ללא תמיכת חומרה מיוחדת מעבר למה שראינו. נבחן את מערכת הקבצים לפי:
 1. **מבנה הנתונים** – איך מערכת הקבצים מארגנת את המידע על הדיסק?
 2. **אופני הגישה** – איך קריאות המערכת () open(), read(), write() פועלות על מבני הנתונים הללו?

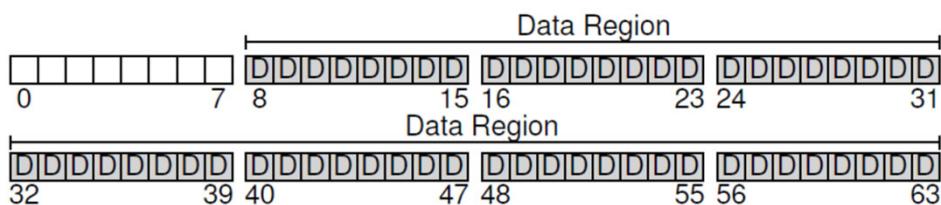
שלב 1: חלוקה לבלוקים

- כאמור, למרות שהדיסק פועל ביחידות של סקטורים (512 בתים),
מערכת ההפעלה מנהלת קבצים ביחידות של בלוקים (4096 בתים).
- לצורך הדוגמה נניח כי הדיסק מכיל 64 בלוקים בלבד.
- ◀ הדיסק בגודל 256KB.



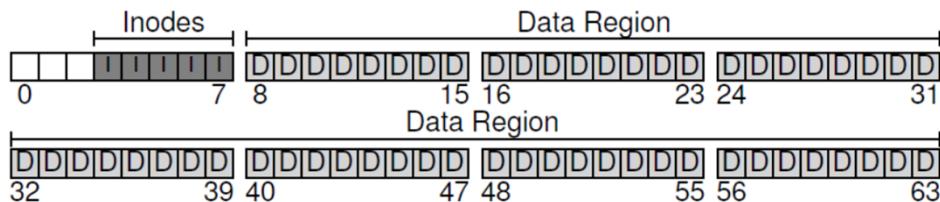
שלב 2: איזור הנתונים (data region)

- מה צריך לשמר במערכת הקבצים?
- הדבר הראשון הוא, כמובן, הנתונים של המשתמשים.
- במערכת הקבצים שלנו נניח כי האיזור המוקדש לנ נתונים של המשתמשים (data region) הוא 56 בלוקים.
- כל מערכת קבצים תשאף כמובן להקדיש חלק גדול ככל הניתן מהdisk לאיזור הנתונים.



שלב 3: טבלת ה-inodes

- כאמור, מערכת הפעלה צריכה לשמר גם inode לכל קובץ.
- ה-node שומר אינפורמציה נוספת (metadata) על כל קובץ, למשל גודל הקובץ, הרשות גישה, חותמות זמן, ועוד.
- כל inode נשמרים במערך רציף בגודל 5 בלוקים בDISK.
- אם נניח כי גודל inode הוא 128 בתים, מה מספר הקבצים המỖימלי האפשרי במערכת הקבצים VSFS?



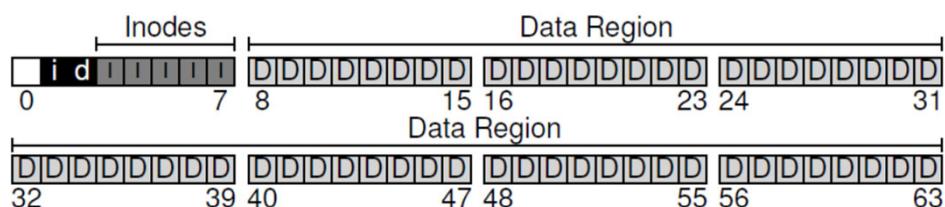
Answer: $5 \times 4KB / 128B = 160$ inodes.

There cannot be more files than inodes.

שלב 4: מערכי ביטים (bitmaps)

- מערכת הפעלה צריכה כموון לעקב אחר הבלוקים הפנויים באיזור הנתונים ואחר ה-nodes הפנויים בטבלה.
 - VSFS משתמש בשני מערבי ביטים (bitmaps) פשוטים כדי לסמן האם האובייקט המתאים פניו (0) או תפוא (1).
 - לכל מערך ביטים נקדים בלוק אחד בדיסק.

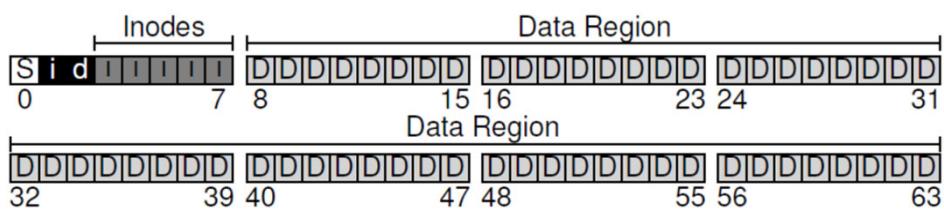
האם זה מספיק?



Answer: yes. In fact, it is a bit of overkill to use an entire 4KB block for these bitmaps; such a bitmap can track whether 32K objects are allocated, and yet we only have 160 inodes and 56 data blocks. However, VSFS uses an entire 4-KB block for each of these bitmaps for simplicity.

שלב 5: סופרבლוק (superblock)

- הבלוק הראשון בדיסק יקרא הסופרבולוק, והוא ישמור מידע כללי על מערכת הקבצים, למשל:
 - מה מספר הבלוקים שלו? מה מספר ה-inodes שלו?
 - היכן מתחילה טבלת ה-inodes?
 - היכן נמצא inode של תיקייה השורש "/"?
- בעת הרכבה של מערכת הקבצים, מערכת ההפעלה תקרא את הסופרבולוק לזכרון וכך תדע כיצד לגשת לקבצים.



שאלה: לרוב נשמר עותקים רזרביים של הסופרבולוק ברחבי הדיסק. למה נהגים לעשות זאת?

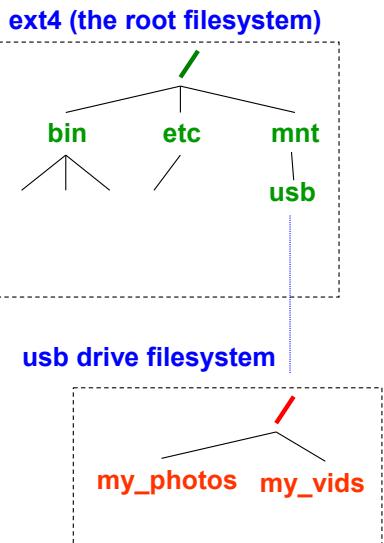
תשובה: בשביל יתרות ועמידות בפני שגיאות. הסופרבולוק הוא מבנה קריטי למערכת הקבצים. בטעדי – לא יוכל לקרוא ממנו דבר או לעשות לה mount.

הרכבה וניתוק של מערכות קבצים

- לינוקס מאפשרת להרכיב (mount) מערכות קבצים שונות **להיררכיה אחת בצוות עץ**.
- פעולת ההרכבה נעשית על-גבי תיקייה קיימת (לרוב ריקה) במערכת הקבצים הנקראת נקודת ההרכבה (mount point).
- לאחר ההרכבה לא ניתן לגשת לקבצים ולתיקיות שהו תחת תיקיה זו לפני ההרכבה.
- לאחר הניתוק ניתן יהיה לגשת שוב לקבצים ולתיקיות שהו תחת נקודת ההרכבה לפני ההרכבה.
- אם הרכבנו כמה פעמים, אז בכל ניתוק נחזיר למערכת הקבצים שהייתה לפני הניתוק.
- הרכבה וניתוק של מערכת קבצים נעשות באמצעות קריאות המערכת (mount() ו-unmount()) ודרשות הרשות מתאימות.

בלינוקס ניתן להרכיב למערכת הקבצים הכללית את אותה מערכת קבצים פיזית מספר פעמים בנקודות ההרכבה שונות.
זה מאפשר להגיע לאותה מערכת קבצים במסלולי גישה שונים (path).

דוגמה: הרכבה של מערכת קבצים



- שורש העץ הוא התיקייה "/"
השייכת למערכת הקבצים
.ext4.
- נרצה להרכיב את מערכת
הקבצים שיושבת בהתקן חיצוני
סוג disk-on-key.
- תיקיית ההרכבה /mnt/usb
תשמש כתיקיית השורש של
מערכת הקבצים החדשה.
- לאחר ההרכבה ניתן לגשת
למידע של התקן בצורה
פומטת, למשל ע"י גישה ל-
. /mnt/usb/my_photos

מציאת inode לפי מסטרן

- בהתנתק מסטרן מסטרן ניתן לחשב בדיק היכן הוא נמצא על הדיסק.
- ספציפית, הסקטור בו נמצא ה-inode מחושב באמצעות:

$$\text{inodeAddr} = \text{inodeTableAddr} + (\text{inodeNumber} \times \text{inodeSize})$$

|-----in bytes-----| |--in bytes--|

$$\text{sector} = \text{inodeAddr} / \text{sectorSize}$$

- לאחר שמערכת הפעלה מצאה את ה-inode, יש בידיה את כל המידע כדי להמשיך ולקראא את הקובץ.

ext2 inode

Size	Name	What is this inode field for?
2	mode	can this file be read/written/executed?
2	uid	who owns this file?
4	size	how many bytes are in this file?
4	time	what time was this file last accessed?
4	ctime	what time was this file created?
4	mtime	what time was this file last modified?
4	dtime	what time was this inode deleted?
2	gid	which group does this file belong to?
2	links_count	how many hard links are there to this file?
4	blocks	how many blocks have been allocated to this file?
4	flags	how should ext2 use this inode?
4	osd1	an OS-dependent field
60	block	a set of disk pointers (15 total)
4	generation	file version (used by NFS)
4	file_acl	a new permissions model beyond mode bits
4	dir_acl	called access control lists

This is just a simplified inode structure; ext2 inodes contain more fields.

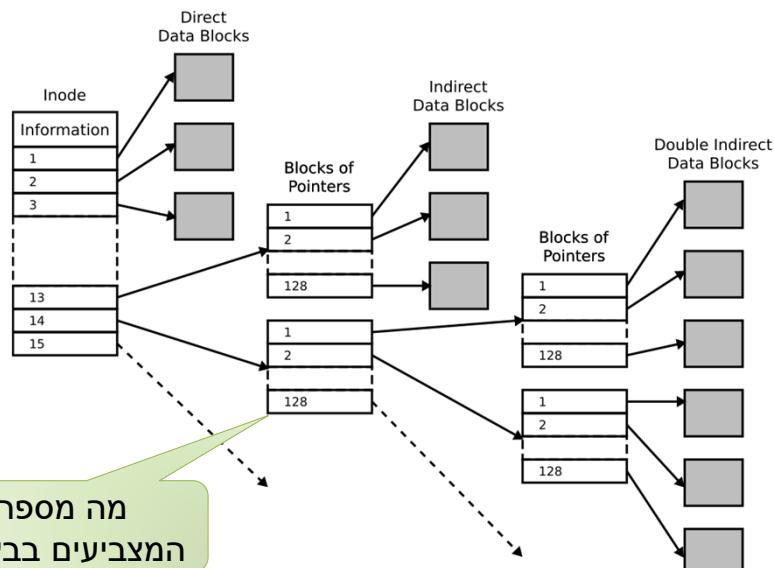
Read more about inode structures in: <https://www.win.tue.nl/~aeb/linux/lk/lk-7.html>

AIR להציג לבLOCKים המרכיבים את הקובץ?

- הדרך פשוטה ביותר היא לשמר בתוך ה-`inode` את המצביעים.
 - מצביע לבLOCK הוא בסך הכל מספר של BLOCK בדיסק.
- דוגמה: `ext2 inode` שומר 12 מצביעים ישירים.
מה המגבלה על גודל הקובץ שנייתן לכנות עם מצביעים ישירים?
 - כדי להתגבר על המגבלה הנ"ל, ניתן להוסיף מצביע לא ישיר – מצביע לבLOCK שיכיל עוד מצביעים ישירים.
 - וגם מצביע לא ישיר כפוי – מצביע לבLOCK שמכיל מצביעים לא ישירים.
 - וגם מצביע לא ישיר משולש – הבנתם את הרעיון...
- את הבLOCKים הנוספים של המצביעים שומרים באיזור הנתונים כי הוא האיזור הגדול ביותר.

Answer: only $12 \times 4KB = 48KB$ can be covered with 12 direct pointers.

מצביעים לבלוקים



תשובה: מספר המצביעים בבלוק אינו בהכרח 128 כי שמויע בשקף, אלא תלוי בגודל הדיסק וגודל הבלוק. נחזור לשאלת הזו בסוף התרגול.

Taken from: https://en.wikipedia.org/wiki/Inode_pointer_structure

אופני הגישה בקריאה מקובץ

- כתעת נניח כי מערכת הקבצים VSFS מורכבת וכי הסופרבלוק כבר נמצא בזיכרון.
- שאר הנתונים (קבצים, תיקיות, `inodes`, ...) עדין בדיסק.
- כמה פעמים ניתן לגשת הקוד הבא לדיסק?

```
int fd = open("/foo/bar", O_RDONLY);
read(fd, buffer, 4096);
read(fd, buffer, 4096);
read(fd, buffer, 4096);
```

מהלך פתיחת+קריאה קובץ

	data bitmap	inode bitmap	root inode	foo inode	bar inode	root data	foo data	bar data[0]	bar data[1]	bar data[2]
open(bar)			read			read				
				read				read		
				read					read	
read()				write					read	
read()				read					read	
read()				write						read

אבחנה: פתיחת קובץ קיימן וקריאה לא ניגשות כלל ל-bitmaps.

שלבי קריית המערכת (open)

- .1. קריית המערכת (open) נדרש צריכה למצוא את ה-eode של הקובץ bar כדי לבדוק הרשות גישה. חיפוש הקובץ מתחילה בתיקיית השורש, אשר ה-eode שלו נמצא במקומ ידוע בדיסק.
- .2. ה-eode של תיקיית השורש מצביע לבLOCKים המרכיבים אותה. מערכת הפעלה תקרא את הבלוקים אלה ותחפש כניסה בשם foo – הכניסה הזאת מצביעה ל-einode של התיקיה foo.
- .3. מערכת הפעלה תקרא את ה-eode של foo כדי לבדוק את הרשות גישה לתיקיה זו.
- .4. מערכת הפעלה תקרא את הבלוקים המרכיבים את התיקיה foo ותחפש בהם כניסה בשם bar.
- .5. בשלב האחרון של (open) יהיה לקרוא את ה-eode של הקובץ bar ולוודא הרשות גישה אליו.

שימוש לב: באלגוריתם המוצג בשקף, FSFS לא מעדכנת את חותמת הזמן של התיקיות לאור המסלול, למרות שמערכת הקבצים "קוראת" את התיקיות אלה במהלך האלגוריתם. מערכות קבצים אחרות יכולות כמובן להתנהג בצורה אחרת בהקשר של עדכון זמן כניסה האחרון לקבצים ותיקיות.

שלבי קריית המערכת (read)

1. קריית המערכת (read) תיגש קודם ל inode של הקובץ bar כדי למצוא את הבלוקים המרכיבים אותו.
2. לאחר מכן מערכת הפעלה תקרא את הבלוק המבוקש מהדיסק.
3. לבסוף מערכת הפעלה תכתוב ל inode כדי לעדכן את חותמת הזמן של הגישה الأخيرة לקובץ.

אופני הגישה בכתיבה לקובץ

- שוב נניח כי מערכת הקבצים VSFS מורכבת וכי הטעורבלוק כבר נמצא בזיכרון.
- שאר הנתונים (קבצים, תיקיות, `inodes`, ...) עדין בדיסק.
- כמה פעמים ניתן לחשוף הקוד הבא לדיסק?

```
int fd = open("/foo/bar",
              O_CREAT | O_WRONLY | O_TRUNC);
write(fd, buffer, 4096);
write(fd, buffer, 4096);
```

מהלך ייצור+כתיבה קובץ

	data bitmap	inode bitmap	root inode	foo inode	bar inode	root data	foo data	bar data[0]	bar data[1]	bar data[2]
create (/foo/bar)		read write	read	read		read	read			
					read write		write			
write()	read write				read			write		
write()	read write				write read				write	
					write					

למה קריאה
ואז כתיבה?

תשובה: בשקף הבא.

שלבי ייצרת קובץ חדש

- יצירת קובץ חדש דומה לפתיחת קובץ קיים בתוספת שלבים הבאים:
- .1. קריית ה-node bitmap כדי למצוא inode פנוי לקובץ החדש.
 - .2. כתיבה חזרה ל-node bitmap כדי לסמן את ה-node בתפואו.
 - .3. איתחול ה-node של הקובץ החדש באמצעות קריאה+כתיבה: ה-node קטן יותר מסקטור שלם (128 בטים לעומת 512 בטים) וכן כדי לאתחל אותו ציריך לקרוא את כל הסקטור המכיל את ה-node ואז לכתוב אותו חזרה לדיסק.
 - .4. כתיבה לבlokים המכילים את התקינה סוף כדי להוסיף את הכניסה המתאימה לקובץ החדש bar.
 - .5. כתיבה ל-node של התקינה סוף כדי לעדכן את חותמות הזמן שלה.

שלבי קריית המערכת write()

- מערכת הפעלה צריכה כמובן לכתוב את הבלוק המבוקש לדיסק.

אבל אם צריך להקצות בלוק חדש לקובץ אז גם:

- .1. קריית ה-bitmap data כדי למצאו בלוק פנוי.
- .2. כתיבה חוזרת ל-bitmap data כדי לסמן את הבלוק התפוס.
- .3. קרייה + כתיבה ל-node המתאים כדי לעדכן את המיקום של הבלוק החדש שהוקצה.

inode cache

- ראיינו כי קרייאות המערכת הנפוצות על קבצים דורשות גישות רבות רבות לדיסק כדי לעדכן את מבני הנתונים.
 - לדוגמה: יצירתקובץ חדש ניגשת 10 פעמים לדיסק.
- כדי לצמצם את כמות הגישות לדיסק, לינוקס שומרת מספר מטמוני:
 - מטמון הדפים (page cache) – עבור המידע של איזור הנתונים.
 - מטמון inodes – עבור המידע של טבלת inode.
 - מטמוני נוספים עבור ה-*smaps* או מבני נתונים אחרים של מערכת הקבצים.

FILE ALLOCATION TABLE (FAT)

משפחה FAT

- FAT הוא שם כולל למשפחה של מערכות קבצים הקוריות על שם מבנה הנתונים המרכזי בו הן משתמשות:
 $FAT = \text{file allocation table}$.
- FAT פותחה ע"י חברת מיקרוסופט בשביל מערכת הפעלה DOS.
- בגלל הפשטות הייחוסית שלה, היא אומצה במהרה גם ע"י כוננים חיצוניים (floppy disks, CD-ROM).
- FAT עדין נמצאת בשימוש נרחב בהתקני فلاש (disk-on-key).
- אנחנו נלמד עליה בתור דוגמה נוספת למימוש של מערכת קבצים.

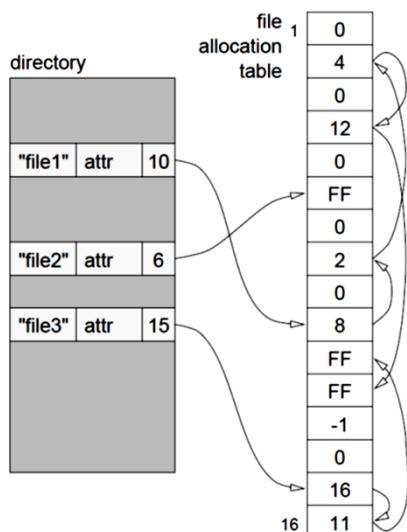
מבנה הדיסק

- VSFS שומרת מצביעים לבלוקים של כל קובץ בתוך ה-inode שלו.
- לעומת זאת, מערכת הקבצים FAT שומרת את הבלוקים בצורה של רשימה מקוشرת.
- כל הרשימות המקוشرת נשמרות בטבלה FAT אשר ממוקמת בתחילת הדיסק. מבנה הדיסק הוא (בערך):

superblock	FAT	data region
------------	-----	-------------

- FAT היא טבלה גלובלית לכל הקבצים אשר "mdbika" את הבלוקים של כל קובץ היחיד אחת.

File Allocation Table (FAT)



- FAT שומרת כניסה לכל בлок בדиск.
- בЛОקים ששיכים לקובץ מסויים מצביעים על הבלוק הבא של אותו קובץ.
- FF מסמן שהו הבלוק האחרון ברשימה (כלומר הבלוק האחרון בקובץ).
- 0 מסמן שהו בлок פנוי.
- -1 מסמן שהו בлок פגום (כדי לא להשתמש בו).

Taken from: "Notes on Operating Systems", by Dror G. Feitelson

תיקיות ב-FAT

- תיקיות ב-FAT הם קבצים רגילים אשר מכילים רשומות מסווג:

filename	metadata	starting block

- כל רשומה מכילה את ה-metadata על הקובץ.
- המשמעות: ה-eodeו נאילו מותמע בתוך הרשימה.
- לכן מערכות קבצים מסווג FAT לא תומכות בקישורים קשיים.

שאלה ממבחן

מועד א', סמסטר חורף תש"פ (2019–2020)

נתוני השאלה

• במערכת קבצים כלשהי, שדומה למערכת הקבצים הקלאסית של UNIX, השדה `arr` ב-`inode` מכיל מערך של מצביעים:

- `inode.arr[0..11]` are direct pointers,
- `inode.arr[12]` is an indirect pointer,
- `inode.arr[13]` is a double indirect pointer,
- `inode.arr[14]` is a triple indirect pointer.

• הניחו שה-`inode` של מערכת הקבצים הינו בגודל 512B (תמיד מושך לכפולה של 512B).

• כמו כן, הניחו שמערכת הקבצים מורכבת (mounted) על דיסק בגודל 1TB בעל גודל בלוק של 8KB.

סעיף א'

- נסמן ב- N את מספר המכביעים שבЛОק מסווג `indirect` יכול להכיל מהו ערך ה- N המקסימלי?
- פתרונות לא נכונים:
- להניח את גודל מכבייע, למשל להניח כי מכבייע הוא 8 בתים בגלן שהארכיתקטורה של המעבד היא 64 בית.
- אין קשר בין מערכת הקבצים למעבד. מערכת הקבצים היא יוצר תוכנות בלבד.
- למצוות את N מתוך המשוואה כי גודל הקובץ המקסימלי הוא 1TB:
$$(12+N^2+N^3)*8KB=1TB$$
- אין קשר בין גודל הקובץ המקסימלי לבין נפח הדיסק.
לדוגמא: קובץ מסוים יכול להיות גדול יותר מנפח הדיסק אם לקובץ יש מספר מכביעים לאוטו בלוק בדיסק (למשל אם הקובץ מורכב מהרבה בלוקים זהים).

פתרון סעיף א'

- מספר הבלוקים בדיסק הוא:
$$1\text{TB} / 8\text{KB} = 2^{40} / 2^{13} = 2^{27}$$
- ← דרישים 27 ביטים לקידוד אינדקס של בלוק.
מספר המצביעים בבלוק של indirect pointers הוא אם כן:
$$8\text{KB} / 27\text{bit} = 2427$$
- שימושו לב להבדל בין בית לבית.
- שימושו לב: השאלה בבקשתה במפורש את N המקיים, ולכן הנחנו כי כל נפח הדיסק מוקדש לבלוקים של נתוניים. גם בהנחה מציאותית יותר שחלק מהdisk (לדוגמה 10%) מוקדש לsuperblock, `inodes`, וכן הלאה - עדין דרישים 27 ביטים לקידוד אינדקס של בלוק, ולכן התשובה נותרת ללא שינוי.

עוד תשובות נכונות

- ניקוד מלא ניתן לסטודנטים שעיגלו את רוחב המצביע (27 ביט) לחזקה שלמה של 2 (כלומר 32 ביט).
- ניקוד מלא ניתן גם לסטודנטים שהוסיפו ביט valid ליד כל מצביע וקיבלו לכך:
$$8\text{KB} / 28\text{bit} = 2340$$
- בפועל, אין צורך בביט valid כי ניתן לקודד invalid pointer בעזרת הצבעה לבлок 0 (שהרי הבלוק הראשון בדיסק מכילס את הסופרבლוק).

הסוף!

או שאולי זו רק ההתחלה...

קורסי המשך

- הנדסת מערכות הפעלה **236376**
- סמינר במערכות מחשבים **236827**
- הגנה במערכות מתוכנות **236350**
- (Reverse Engineering) – הנדסה לאחרור **236653**
- תקשורת באינטרנט **236341**
- מערכות אחסון מידע **236322**
- מאיצים חישוביים ומערכות מואצנות **046278**

הערות וטיפים לבחינה

- הבחינה בחומר סגור.
- לא צריך לזכור הכל.
- כן צריך לזכור את המושגים החשובים (...). (spinlock, TLB, inode, FDT, ...).
- חשוב להקפיד על הבנת הנקרא בשאלות.
- לדוגמה: "תזכורת" לעומת "הגדרה".
- אנחנו משתמשים לנוכח כל שאלה בקפידה אבל לא תמיד מצלחים...
- יש לנמק את כל הפתרונות שלכם – גם בשאלות האמריקאיות!
- אם נתקעים, כדאי להשתמש בשיטת האלימינציה.
- ואפשר להפעיל גם טכניקות של פיזיקאים: ייחדות, גבולות, סימטריה.
- מותר להשתמש במחשבון.
- למרות שלא יהיו חישובים מורכבים.
- למי ששכח מחשבון, אפשר גם להיעזר במריצים/מתרגלים.

דוגמה לניסוח בעיתוי של שאלה: מה היתרון של מכוניות?
הניסוח בעיתוי כי לא ברור מול מה משווים את המכוניות.
אם משווים מול אופננו – אז המכונית בטוחה יותר. אם משווים מול מטו – אז המכונית זולה יותר.